VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Nástroj pre podporu ITIL

Tool for ITIL Processes

2012

Martin Gajdičiar

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Martin Gajdičiar**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Nástroj pro podporu ITIL
Tool for ITIL Processes

Zásady pro vypracování:

Cílem práce je vytvořit, nebo najít již existující, univerzálně použitelné komponenty vyvinuté na technologii JSF 2.0, které bude možné dle potřeby upravovat a využívat v různých systémech. Po studentovi je požadována především implementace obecného seznamu – listu dat, stromové struktury a formuláře stránky do výsledného demonstračního IS.
Důraz je kladen především na praktický výstup a funkčnost demonstračního systému.

1. Nastudujte technologie java pro tvorbu webových aplikací (JSF 2.0, ICEfaces, Spring). Z prostudovaných zvolit nejvhodnější (ne nutně nejnovější) technologii pro implementaci nástroje pro podporu ITIL.
2. Implementované komponenty budou schopné poskytovat rozhraní pro použití jakékoliv databázové vrstvy.
3. Implementaci, postup a návrhy průběžně konzultujte s vedoucím práce.
4. Základem aplikace bude modul uživatelů, rolí, skupin a práv. K těmto bude následně implementován malý incident management modul.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Milan Pohančeník**

Datum zadání: 18.11.2011
Datum odevzdání: 04.05.2012

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

V Ostrave 4. mája 2012 .....

Rád by som na tomto mieste poďakoval vedúcemu mojej práce Ing. Milanovi Pohančeníkovi za cenné odborné rady, čas strávený pri konzultáciách a poskytnuté študijné materiály. Taktiež sa chcem poďakovať svojej priateľke a mojej rodine za podporu počas tvorby tejto práce.

Abstrakt

Práca sa zaoberá vývojom demonštračného informačného systému v programovacom jazyku Java EE. IS je určený na podporu ITIL procesov. Obsahuje správu používateľov, skupín, rolí a práv spolu s incident management a problem management modulom. Hlavnou náplňou vývoja bolo využitie rôznych komponentov založených na technológii JSF 2.0 a demonštrácia ich reálneho nasadenia. Dôraz bol kladený aj na dôležité aspekty IS, ako napríklad bezpečnosť, použiteľnosť akejkoľvek databázovej vrstvy, viac užívateľský prístup a celkovú funkčnosť. Práca obsahuje popis jednotlivých vrstiev IS, ich prepojenie a implementáciu, spolu s popisom použitých komponentov a rôznych problémov.

Kľúčové slová: JSF 2.0, Java EE, Primefaces, ITIL, IS, Service Desk

Abstract

The aim of this thesis is a develompment of information system under the Java EE programming language. The main role of the developed system is to support ITIL processes. IS manages user, group, role and permission administration and contains incident management and problem management module. The main intent during the develompment of this IS was to use different components developed under the JSF 2.0 technology and to demonstrate their usage. Development was focused on important parts of an IS like security, ability to use any database layer, multi-user accessibility and overall functionality. Thesis contains description of IS layers, their logical connection and implementation along with description of used components and different problems.

Keywords: JSF 2.0, Java EE, Primefaces, ITIL, IS, Service Desk

Zoznam použitých skratiek a symbolov

AJAX	–	Asynchronous JavaScript and XML
API	–	Application Programming Interface
AS	–	Aplikačný Server
CSS	–	Cascading Style Sheets
DAO	–	Data Access Object
DB	–	Database
DTO	–	Data Transfer Object
EIS	–	Executive Information System
EJB	–	Enterprise JavaBeans
EL	–	Expression Language
ER	–	Entity-Relationship
GUI	–	Graphical User Interface
HTTP	–	Hypertext Transfer Protocol
HTTPS	–	Hypertext Transfer Protocol Secure
IDE	–	Integrated Development Environment
IM	–	Incident Management
IS	–	Information System
IT	–	Information Technology
ITIL	–	Information Technology Infrastructure Library
ITSM	–	IT Service Management
EE	–	Enterprise Edition
SE	–	Standard Edition
JDBC	–	Java Database Connectivity
JPA	–	Java Persistence API
JSF	–	Java ServerFaces
JSP	–	Java ServerPages
MD5	–	Message-Digest Algorithm
MVC	–	Model–View–Controller
OODBMS	–	Object-Oriented Database Management System
ORM	–	Object-Relational Mapping
PM	–	Problem Management
POJO	–	Plain Old Java Object
RDBMS	–	Relational Database Management System

SD	–	Service Desk
SQL	–	Structured Query Language
UI	–	User Interface
URL	–	Uniform Resource Locator
XHTML	–	Extensible Hypertext Markup Language
XML	–	Extensible Markup Language

Obsah

Zoznam tabuliek

Zoznam obrázkov

Zoznam výpisov zdrojového kódu

1 Úvod

V dnešnej dobe sa vo firmách a podnikoch na každom kroku stretávame s **IT službami**, ktoré podporujú alebo priamo umožňujú fungovanie niektorého biznis procesu, resp. biznis činnosti. Pod pojmom biznis proces rozumieme hlavné, napr. obchodné procesy, ktoré priamo ovplyvňujú ziskovosť a fungovanie organizácie.[22] Riadením služieb informačných technológií sa zaoberá **ITSM**.[21] Popis najlepších skúseností z praxe, ako dosiahnuť efektívne riadenie služieb IT, je popísaný v rade knižných publikácií, ktoré nesú spoločný názov **ITIL**.[23] Podrobnejšie k ITSM a ITIL v sekcii 3.1. Na podporu procesov popísaných v ITIL, ako napr. incident management a problem management sa v súčasnosti vyvíjajú rôzne softwarové riešenia. Takéto softwarové produkty sú dnes neodmysliteľnou súčasťou každej väčšej firmy.

Pôvodne bolo zámerom tejto práce vyvinúť opätovne použiteľné komponenty na technológii `JSF 2.0` pre prezentačnú vrstvu takéhoto systému. Teda napríklad tabuľku ktorá by umožňovala filtrovanie dát, ich zoraďovanie, stránkovanie a pod. V priebehu času [1] sa ale zistilo, že by bolo výhodnejšie sústrediť sa na samotný vývoj takéhoto systému a radšej využiť existujúce komponenty z niektorých **Open Source** projektov.

Cieľom tejto práce je teda nájsť alebo vytvoriť čo najvhodnejšie komponenty na technológii `JSF 2.0` a tie následne zakomponovať do zjednodušenej, avšak funkčnej formy SD s možnosťou správy používateľov, skupín, rolí a práv, spolu s implementovaným incident a problem management modulom podľa ITIL. IS je vytvorený na programovacom jazyku Java EE 6, viď. sekcia 2.1.

V jednotlivých kapitolách popisujem postup implementácie a návrhu dôležitých častí IS, ako napríklad bezpečnosť, použiteľnosť akejkoľvek databázovej vrstvy a celkovú funkčnosť. Práca obsahuje tiež popis jednotlivých vrstiev IS, ich prepojenie a implementáciu, spolu s popisom použitých `JSF 2.0` komponentov a rôznych problémov.

[1] Pred rokom bola práca určená pre firmu ale študent ju nedokončil

2 Použité technológie

V tejto kapitole opíšem technológie používané v mojej práci. Táto kapitola predpokladá základné znalosti programovacieho jazyka Java a `HTTP` komunikácie.

2.1 Java EE 6

Java EE (kedysi označovaná ako Java 2 Enterprise Edition alebo J2EE) je súčasťou platformy Java určená pre vývoj a chod podnikových aplikácií a informačných systémov. Java EE nadväzuje na Javu SE.

Java EE poskytuje bohaté API pre vývoj webových aplikácií. Tieto aplikácie potom fungujú na aplikačných serveroch. Tie sú dodávané rôznymi dodávateľmi. Aplikácia by mala byť teoreticky práce schopná na ktoromkoľvek AS ktoréhokoľvek dodávateľa implementujúceho príslušnú verziu špecifikácie - koncept prenositeľnosti. Väčšina AS však dopĺňa niektoré vlastnosti nad rámec špecifikácie a aplikácia využívajúca týchto vlastností potom nie je prenositeľná. Zoznam niektorých aplikačných serverov :

- Open Source :
  - GlassFish
  - JBoss
  - Apache Tomcat (čiastočná implementácia)
- Komerčné :
  - IBM WebSphere
  - BEA WebLogic

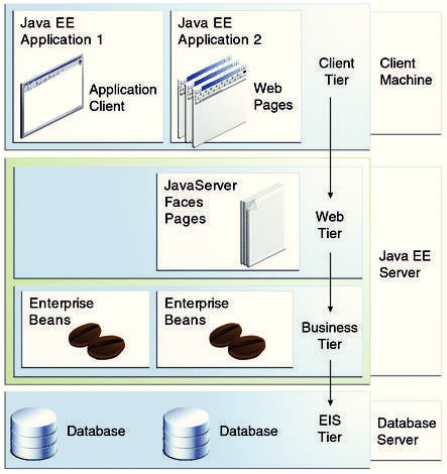
Platforma Java EE 6 podobne, ako jej predchodca ver. 5 využíva anotácie zavedené v Jave SE, ktoré poskytujú zjednodušený model programovania. Využívanie anotácií eliminuje potrebu používania `deployment descriptorov`, hoci ich použitie nevylučuje, tým že dovolí programátorovi umiestniť konfiguračné dáta priamo do časti zdrojového kódu. Týmto spôsobom sa všetky informácie, ktoré sú vyžadované Java EE serverom k úspešnému načítaniu a konfigurácii komponentu nachádzajú na jednom mieste, čo veľmi uľahčuje ich spravovanie. Java EE 6 taktiež podporuje `Dependency Injection (inversion of control)`, tzn., že Java EE komponenty môžu používať anotácie na získanie zdrojov, ako napríklad pripojenie k DB, ktoré sú automaticky injektované Java EE kontajnerom pri deploy alebo spustení aplikácie. Týmto spôsobom ukryjeme `lookup` a vytvorenie požadovaných zdrojov pred developerom a zredukujeme množstvo kódu. Tento postup môže byť použitý v `EJB` kontajneroch, web kontajneroch a v klientoch aplikácie. Viac k architektúre Java EE 6 aplikácie v nasledujúcej časti.[14]

2.1.1 Architektúra Java EE aplikácie

Platforma Java EE používa distribuovaný viacúrovňový aplikačný model pre enterprise aplikácie. Aplikačná logika je rozdelená do komponentov podľa svojej funkcie. Tieto komponenty sú ďalej zaradené do jednotlivých vrstiev Java EE aplikácie.

Obr.1 ukazuje dve viacvrstvové Java EE aplikácie rozdelené do nasledujúcich vrstiev :

- Komponenty klientskej vrstvy bežia na stroji klienta resp. používateľa.
- Komponenty webovej vrstvy bežia na Java EE serveri.
- Komponenty biznis vrstvy bežia na Java EE serveri.
- Software EIS vrstvy beží na EIS serveri.



Obr. 1: Viacvrstvové aplikácie

Java EE aplikácie sú vo všeobecnosti považované za trojvrstvové, pretože sú distribuované cez 3 polohy: stroj klienta, stroj na ktorom beží Java EE server a stroje na ktorých je umiestnená databáza.[5]

JavaBeans

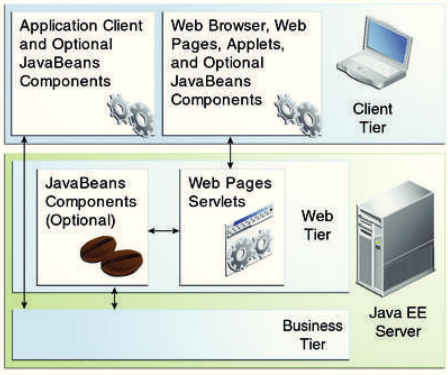
Vrstva klienta a serveru môže tiež obsahovať komponenty založené na `JavaBean` architektúre (`JavaBean` komponenty) na spravovanie dátového toku medzi nasledujúcimi:

- Aplikačným klientom alebo appletom a komponentmi bežiacimi na Java EE serveri
- Komponentmi aplikačného serveru a databázou

K jednotlivým atribútom (properties) `JavaBeany` pristupujeme pomocou definovaných get a set metód.[5]

Web Components

Java EE web komponenty sú buď Servlety alebo webové stránky vytvorené použitím technológie JavaServer Faces a/alebo JavaServerPages technológie (JSP stránky). Servlety sú triedy programovacieho jazyka Java, ktoré dynamicky spracúvajú požiadavky a vytvárajú odpovede, bližšie viď. 2.1.3. JSP stránky sú textovo založené dokumenty, ktoré pracujú ako Servlet ale umožňujú prirodzenejší postup vytvárania statického obsahu. Technológia JavaServer Faces je postavená na technológii Servletov a JSP [2] a poskytuje framework pre tvorbu komponentov používateľského rozhrania pre webové aplikácie. Viac o JSF v sekcii 2.2.[5]



Obr. 2: Webová vrstva a Java EE aplikácie

Business Components

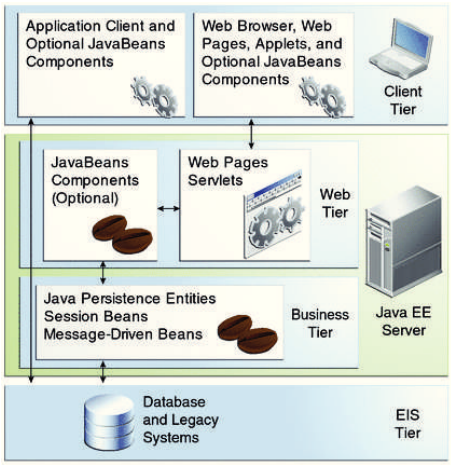
Biznis logika je spravovaná Enterprise JavaBeanami (EJB), ktoré bežia buď na biznis vrstve alebo webovej vrstve. Obr. 3 ukazuje ako Enterprise Bean príma dáta od klienta, spracuje ich, a posiela ich EIS vrstve napríklad na uloženie. Enterprise Bean taktiež získava dáta z dátového zdroja, napr. aj prostredníctvom DAO tried, ak je to potrebné tak ich spracuje a pošle ich späť klientovi. Bližšie sú EJB popísané v sekcii 2.1.2[5]

Java Database Connectivity API

Java Database Connectivity (JDBC) API nám dovoľuje volať SQL príkazy z metód naprogramovaných v jazyku Java. JDBC API používame na prístup do do databáze najčastejšie z EJB alebo DAO triedy - možné, no nie príliš vhodné je tiež pristupovať do DB priamo z JSP stránky alebo Servletu. JDBC API má 2 časti:

- Rozhranie na úrovni aplikácie používané aplikačnými komponentmi na prístup do DB
- Rozhranie na poskytovanie služby na pripojenie JDBC driveru k Java EE platforme

[2] JSP stránky sú od verzie JSF 2.0 nahradené Faceletami



Obr. 3: Business a EIS vrstva

JDBC tvorí najnižšiu úroveň prístupu k DB v Jave. V súčasnej dobe sú tiež dostupné rôzne persistence frameworky, ako **Hibernate, JPA, či iBATIS**, ktoré využívajú JDBC.[13]

2.1.2 Enterprise JavaBean technológia

EJB môžeme považovať za stavebný blok, ktorý môže byť použitý samostatne, alebo spolu s inými EJB na vykonávanie biznis logiky na Java EE serveri, ako som už spomenul. EJB sú buď Session Beany alebo Message-driven Beany.[10]

Session Beany

Session Bean je opätovne použiteľný komponent, ktorý zapuzdruje biznis logiku enterprise aplikácie. Klient pristupuje k Session Beanam, ktoré sa nachádzajú na aplikačnom serveri a volá metódy, ktoré uskutočňujú určité akcie v AS.

Existujú dva typy Session Bean:

- **stateful**
- **stateless**

Stateful Session Beany sa používajú na reprezentáciu stavu session klienta. Tento typ Bean je vhodný ak stav reprezentuje interakciu medzi klientom a danou Session Bean. Stateful Session Bean môže tiež uchovávať informácie o klientovi medzi jednotlivými volaniami metód. Ak klient prestane používať túto Bean tak je zrušená a informácie o stave sa stratia.

Stateless Session Bean sa naopak vôbec nestará o stav klienta. Tým pádom dokáže táto Bean obsluhovať viacerých klientov, nie sú potrebné žiadne dáta o klientovi. Stav Stateless Session Bean je zdieľaný medzi všetkými klientmi, ktorí ju používajú. Jedine tento typ EJB dokáže implementovať webové služby (web services).[11]

Message-Driven Bean

Táto Bean asynchrónne prijíma a spracúva správy zasielané Java EE komponentmi alebo inými aplikáciami pomocou Java Message Service. Message-driven Beany sú podobné Stateless Session Beanam, pretože nepracujú s dátami špecifickými pre klienta. Tým pádom si viaceré ekvivalentné Message-driven Beany môžu konkurovať pri spracovaní prichádzajúcich správ. Klient Message-driven Beany s ňou nekomunikuje priamo tým, že volá jej metódy. Namiesto toho klient zašle správu do cieľa, ktorý je spracovávaný touto Beanou. Správa sa vybaví okamžite po príchode s pomocou onMessage metódy Beany.[12]

2.1.3 Java Servlet technológia

Java Servlet technológia umožňuje definovať Servlet triedy spracujúce HTTP požiadavky (ďalej request). Hoci Servlety môžu odpovedať na akýkoľvek typ requestu, bežne sa používajú na rozšírenie možností aplikácie hostovanej na web serveru.

V čase deploy aplikácie, je každý Servlet namapovaný na určitú URL adresu, alebo vzor URL adries špecifikovaný v deployment descriptore aplikácie, web.xml (môžeme samozrejme použiť aj anotácie). Ak sa prichádzajúci request zhoduje s niektorým vzorom, tak prislúchajúci Servlet spracuje túto požiadavku. Ak zatiaľ neexistuje žiadna inštancia Servletu vo web kontajneri, tak sa vytvorí a inicializuje pred spracovaním požiadavky. [8]

Filter

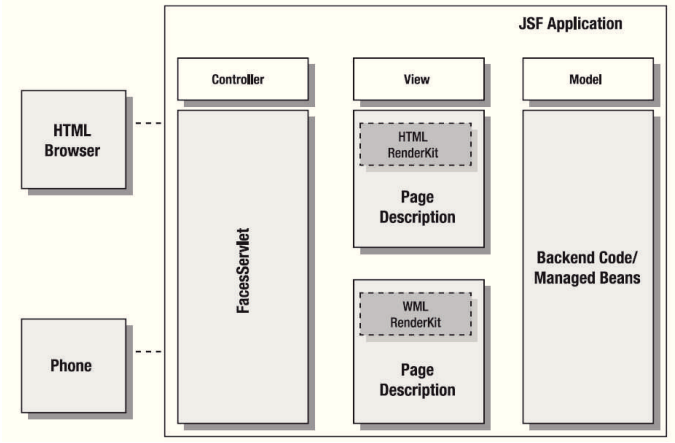
Filter je trieda, ktorá dokáže preskúmať a modifikovať request predtým než je spracovaný web komponentom. Potom, čo komponent vygeneruje response, môže Filter preskúmať a modifikovať aj túto response. Filtre môžu ovplyvniť spôsob akým je request spracovaný. URL, resp. request môže byť filtrovaný viacerými Filtrami, tzv. filter chain. Mimo to môže byť Filter namapovaný na viaceré URL vzory adries. Pred tým, než komponent spracuje request sa vykonajú všetky priradené Filtre v takom poradí v akom boli špecifikované. To isté v opačnom poradí sa deje aj pre response. Najdôležitejšia metóda Filtra je doFilter. V nej spracúvame request/response.

Filtre sa väčšinou používajú na autentizáciu, šifrovanie, prevod XML a rôzne iné problémy. Umožňujú nám meniť requesty a responses, podľa toho, čo potrebujeme. [9]

2.2 JSF

JavaServer Faces je komponent framework na strane serveru určený na zjednodušenie vytvárania používateľských rozhraní pre webové aplikácie postavené na Jave. JSF poskytuje flexibilný model na renderovanie komponentov v rozličných druhoch HTML alebo iných značkovacích jazykoch a technológiách. Objekt Renderer generuje značky na vykreslenie komponentov a konvertuje dáta uložené v modely na typy, ktoré môžu byť

reprezentované vo `view`. `JSF` je založené na architektúre MVC, viď. obr.4 Okrem toho nám `JSF` poskytuje, čo je najhlavnejšie, aj množstvo GUI komponentov.



Obr. 4: MVC architektúra s JSF

Nasledujúce funkcie dopĺňajú samotné GUI komponenty:

- Validácia vstupov
- Správa udalostí
- Prevod dát medzi objektmi modelu a komponentmi
- Vytváranie managed model objektov
- Konfigurácia navigácie medzi stránkami
- `Expression Language` (`EL`)

`Expression Language` je mechanizmus na prístup a modifikáciu dát uložených v `JavaBean` komponentoch, ďalej na volanie **public** a **static** metód a taktiež na vykonávanie aritmetických operácií a logických porovnávaní. V spojení s `JSP` stránkami sú umožnené iba read-only výrazy. Vo `Faceletoch` (`JSF 2.0`) je možné vykonávať read aj write výrazy.[6][2][1]

```
//JSP
${fooBean.fooProperty} // fooProperty zastupuje getter – getFooProperty();

//Facelet
#{fooBean.fooProperty} // fooProperty zastupuje getter aj  setter
```

Výpis 1: Ukážka EL

2.2.1 JSF 2.0

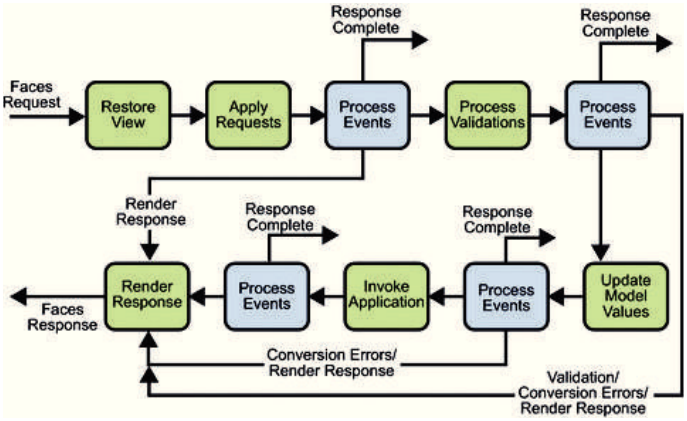
V Java EE 6 boli do JSF pridané nové funkcie:

- Možnosť používať anotácie namiesto konfiguračných súborov na špecifikáciu Managed Bean a iných komponentov (napr. @ManagedBean).
- Snaha o odstránenie faces-config.xml, avšak v niektorých prípadoch je stále potrebné.
- Facelety, nahradili JavaServer Pages (JSP). Facelety používajú XHTML súbory.
- Podpora AJAX
- Kompozitné komponenty
- Implicitná navigácia
- @ViewScoped Beany

S JSF 2.0, sa tiež zrodili krajšie vypadajúce a flexibilnejšie knižnice komponentov, mimo iné aj PrimeFaces a OpenFaces.[7][3]

Životný cyklus JSF 2.0

Pre pochopenie ako JSF funguje je treba dobre poznať jeho životný cyklus viď. obr. 5 , ktorý sa skladá z nasledujúcich fáz:



Obr. 5: Životný cyklus JSF

Restore view - fáza 1

Controller JSF frameworku využije view ID, aby vyhľadal všetky komponenty súvisiace s týmto view. Ak view neexistuje, tak ho JSF controller vytvorí. Naopak ak view už existuje tak JSF controller použije toto view. View obsahuje všetky GUI komponenty.

V tejto fázy životného cyklu sú prítomné tri druhy view:

- **new** view - nové view
- **initial** view- inicializačné view (prvé načítanie stránky)
- **postback** - odpoveď zo strany klienta (napr. po stlačení tlačidla)

Každé z nich sa spracúva inak.

V prípade **new** view, JSF vytvorí view z Faceletu a priradí komponentom príslušných správcov udalostí a validátory. View sa uloží do objektu FacesContext. FacesContext objekt obsahuje všetky informácie o stave komponentov, pomocou ktorých dokáže JSF spravovať ich stav pre aktuálny request v aktuálnej session. FacesContext uchováva aktuálne view v atribúte viewRoot.

V prípade **initial** view JSF vytvorí prázdne view. Toto view bude naplnené hneď, ako používateľ začne vykonávať nejaké akcie. Z initial view JSF pristupuje priamo k fáze **render response**.

V prípade **postback**, view k danej stránke už existuje, takže ho treba len obnoviť. Ďalšou fázou po **postbacku** je fáza **apply request values**.

Apply request values - fáza 2

Potom, čo je vybudovaný strom komponentov, každý komponent si získa svoju novú hodnotu z parametrov requestu za použitia svojej decode metódy. Hodnota sa potom uloží lokálne do komponentu. Ak zlyhá konvertovanie hodnoty, vygeneruje sa chybové hlásenie asociované s týmto komponentom a uloží sa do fronty chybových hlásení vo FacesContext. Toto chybové hlásenie sa potom zobrazí počas fázy **render response**, spolu s inými chybami validácie, ktoré nastali pri vykonávaní fázy **process validation**.

Process validation - fáza 3

Počas tejto fázy, controller JSF spustí a vykoná všetky validátori , ktoré sú registrované v strome komponentov. Validátor zistí aké pravidlá pre validáciu boli zadané (túto informáciu nájde v atribútoch komponentu, ktoré tieto pravidlá špecifikujú) a porovná tieto pravidlá s hodnotou, ktorá bola lokálne uložená pre tento komponent.

Update model values - fáza 4

Potom, čo JSF controller rozhodne, či sú dáta validné môže prejsť stromom komponentov a nastaviť korešpondujúce atribúty objektov na strane serveru do lokálnych

hodnôt daného komponentu. JSF controller aktualizuje iba také property Beany na ktoré je ukázané atribútom value nejakého vstupného(input) komponentu, napr.:

<h:inputText value="#{fooBean.fooProperty}">.

Ak lokálne dáta komponentu nemôžu byť prevedené na typy špecifikované jednotlivými property danej Bean, životný cyklus JSF prejde priamo na fázu **render response**, takže stránka sa vykreslí a zobrazia sa chybové hlásenia.

Invoke application - fáza 5

Počas tejto fázy, sa JSF controller postará o udalosti na úrovni aplikácie, ako je napríklad odoslanie formulára alebo odkázanie sa na inú stránku. Hodnoty komponentov budú v tomto čase skonvertované, zvalidované a aplikované na objekty modelu, takže ich môžeme používať pri vykonávaní aplikačnej biznis logiky.

Render response - fáza 6

Posledná fáza zobrazí view so všetkými jeho komponentmi s ich aktuálnym stavom. [17][18][19]

2.2.2 Managed Beany a JSF 2.0

V JSF 2.0 môžeme Managed Beany vytvárať pomocou anotácie @ManagedBean a špecifikovať názov tejto Bean, pod ktorým bude dostupná v EL pomocou atribútu name. Managed Bean potom nie je potrebné registrovať vo faces-config.xml.

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean(name="fooBean")
@RequestScoped
public class Bean {
    // ...
}
```

Výpis 2: Vytvorenie Managed Bean pomocou anotácie

Teraz bude táto Managed Bean dostupná v EL takto: #{fooBean}. Ak by sme nezadali atribút name tak by JSF implicitne pristupovalo k tejto Bean štýlom #{názovTriedy}. Treba si všimnúť, že JSF samo prevedie prvé písmeno názvu triedy na lower-case.

Bean scopes

Bean scope definuje kedy má byť Managed Bean vytvorená a kedy zničená, v podstate sa jedná o jej trvanie.

JSF 2.0 poskytuje 6 preddefinovaných @ManagedBean scopes. Tu je ich popis životného cyklu a použitia:

- @RequestScoped: Bean v tomto scope prežíva pokiaľ existuje aj súvisiaci HTTP request-response. Je vytvorená na HTTP request, ktorý ju zahŕňa a zničená keď sa dokončí HTTP response asociovaná k danému HTTP requestu (rovnako to platí aj pre AJAX requesty). JSF uchováva Bean ako atribút HttpServletRequest s tým, že meno Managed Bean vystupuje ako kľúč. Je dostupná aj cez ExternalContext.getRequestMap(). Tento scope používame čisto pre request-scoped dáta. Napríklad pre staré známe GET requesty, ktoré by mali prezentovať nejaké dynamické dáta koncovému používateľovi závisiac od parametrov. Tento scope je tiež možné použiť pre jednoduché non-AJAX formuláre, ktoré nepotrebujú meniť stav modelu počas vykonávania.

- @ViewScoped: Bean v tomto scope žije tak dlho pokiaľ používateľ pracuje s rovnakým JSF view v okne prehliadača. Vytvára sa na prvý HTTP request obsahujúci túto Bean a je zničená na postback, ktorý odkazuje na iné view. JSF ukladá Beany tohoto typu do UIViewRoot.getViewMap() s menom Managed Bean ako kľúčom, ktorý sa uloží do session. Tento scope používame pre komplexnejšie formuláre, ktoré používajú AJAX, ďalej pre view, ktoré obsahujú dátové tabuľky a pod.

- @SessionScoped: Bean v tomto scope žije pokiaľ existuje aj HTTP session. Vytvára sa na prvý request, ktorý obsahuje túto Bean. Bean je následne uložená do session a odstránená je v momente keď je session zrušená (alebo keď manuálne Bean odstránime zo session map). JSF uchováva Bean ako atribút HttpSession s menom Managed Bean ako kľúčom. Je tiež dostupná z ExternalContext.getSessionMap(). Tento scope používame čisto pre session-scoped dáta, teda také, ktoré môžu byť bezpečne zdieľané medzi všetkými view rovnakej session. Napr. si môžeme takto uchovávať prihláseného používateľa.

- @ApplicationScoped: Bean v tomto scope žije počas behu celej aplikácie. Vytvára sa na prvý request, ktorý zahŕňa túto Bean, alebo pri štarte samotnej aplikácie ak je atribút eager u @ManagedBean nastavený na **true**. Táto Bean je zničená až pri ukončení celej web aplikácie alebo keď ju manuálne odstránime z application map. JSF uchováva túto Bean ako atribút ServletContext s názvom Managed Bean ako kľúčom. Dostupná je tiež cez ExternalContext.getApplicationMap(). Tento scope používame čisto pre dáta aplikácie, ktoré môžu byť zdieľané medzi všetkými session. Takáto Bean sa často nazýva support Bean.

- @NoneScoped: Bean v tomto scope žije počas vyhodnocovania EL príkazu. Vytvorí sa na začiatku vyhodnocovania EL a je zničená okamžite po jeho ukončení. Táto Bean sa nikde neukladá. Tento scope používame len pre Beany, ktoré sú určené

na to, aby boli injektnuté do inej Bean iného scope. Injektnutá Bean bude potom žiť pokiaľ bude žiť aj Bean do ktorej je injektovaná (acceptor Bean).

- @CustomScoped: Bean v tomto scope žije tak dlho, ako jej vstup vo vytvorenej Map, ktorá je vytvorená pre účel tohoto scope. Túto Map musíme vytvoriť a pripraviť samy v širšom scope, napr. v session scope. Odstraňovanie tejto Bean z Map musíme riadiť tiež samy. Tento typ scope používame, ak žiaden z daných scopes nevyhovuje našim požiadavkám.

Výber správneho scope pre Bean je veľmi dôležitá časť implementácie. Zabránime tým mnohým problémom a neefektívnemu využívaniu pamäte a dátovej linky. Ak Bean obsahuje mix request-scoped a session-scoped dát, mali by sme ju rozdeliť na viaceré Beany s rôznymi scopes, ktoré budú spolupracovať - injektneme jednu Bean do druhej, viz. ďalej. [15] [3] [16]

Injektovanie Managed Bean

Na injektnutie jednej Managed Bean do druhej používame @ManagedProperty. Injektovanie je obzvlášť užitočné ak máme @Request alebo @View scoped Bean asociovanú s určitou stránkou a chceme mať okamžitý prístup k @Session alebo @Aplication scoped Bean. Napríklad, aby sme získali aktuálne prihláseného používateľa, ktorého si uchovávame v @Session scoped Managed Bean.[15]

2.2.3 PrimeFaces

Ako zdroj komponentov som zvolil, podľa môjho názoru, momentálne najlepší JSF komponent framework na trhu: PrimeFaces. Má širokú používateľskú základňu, kompletnú podporu pre JSF 2.0, AJAX, jQuery, rapídny vývoj, dobre spracovanú používateľskú príručku, množstvo použiteľných komponentov a taktiež je integrovaný s jQuery UI Themeroller CSS frameworkom. Dokonca konkurenčný framework IceFaces je založený na PrimeFaces. [20]

Na to, aby sme mohli v aplikácii použiť Primefaces stačí len pridať k projektu primefaces.jar a vo faceletoch deklarovať menný priestor xmlns:p="http :// primefaces.org/ui".

Pre použitie niektorej z preddefinovaných tém stačí do web.xml pridať nový <context −param>.

```
<context−param>
   <param−name>primefaces.THEME</param−name>
   <param−value>redmond</param−value>
</context−param>
```

Výpis 3: Pridanie Primefaces témy do web.xml

3 Teoretické východiská

3.1 ITSM a ITIL

ITSM

Pri riadení IT organizácie treba definovať a popísať služby, ktoré úsek IT poskytuje zamestnancom, naučiť zamestnancov s týmito službami pracovať v kontexte ich každodenných činností a následne tieto služby kontinuálne riadiť, a to ako na operatívnej, tak aj na strategickej úrovni. Disciplína, ktorá sa tejto oblasti venuje, sa volá **IT Service Management (ITSM)**, po slovensky: riadenie služieb informačných technológií. Popis najlepších skúseností z praxe, ako dosiahnuť efektívne riadenie služieb IT, je popísaný v rade knižných publikácií, ktoré nesú spoločný názov **ITIL**.[21]

ITIL

ITIL je súbor knižných publikácií, ktorý obsahuje zbierku najlepších skúseností z odboru riadenia služieb informačných technológií. Tieto publikácie sú vo vlastníctve britskej vlády, konkrétne Cabinet Office (Úrad vlády Jej veličenstva), ktorá okrem ITIL dohliada ešte na šesť ďalších celosvetovo používaných manažérskych rámcov a metodík. Názov ITIL vznikol ako skratka slov **Information Technology Infrastructure Library**.[23]

ITIL nie je predpis, ani norma

Predpisov v knižnici nájdeme iba zopár. Zväčša ide o odporúčania, a z toho plynie základné pravidlo práce s ITIL, ktoré znie v angličtine **adopt and adapt**, a ktoré môžeme parafrázovať nasledovne: zoznámte sa s tým, aké sú najlepšie skúsenosti v odbore ITSM, a na základe týchto skúseností si vytvorte svoj vlastný systém riadenia služieb, ktorý bude vyhovovať Vášmu prostrediu. V ITIL je len veľmi málo imperatívov, tzn. **ITIL málokedy niečo striktne predpisuje**. Jeden príklad za všetky: Podľa ITIL by procesy Incident management a Problem management nemali byť nikdy zlúčené do jedného procesu, a taktiež manažéri týchto dvoch procesov by mali byť dve rôzne osoby.[24]

ITIL je rámec ITSM, nie metodika ITSM

ITIL nepopisuje detailne jednotlivé prvky best practice. Poskytuje len rámcové návody, nie podrobné opisy toho, ako reagovať na každú špecifickú situáciu, resp. ako riešiť podporu služieb v špecifických podmienkach. ITIL napr. vraví, že všetky incidenty majú byť prioritizované s ohľadom na dopad, ktorý má ich existencia na biznis, a naliehanie používateľa, požadujúceho jeho odstránenie, čiže podľa aktuálnej potreby pracovať so službou, na ktorej incident vznikol. ITIL nepredpisuje, koľko má byť úrovní pre dopad a naliehanie používateľa, ani aké majú byť výsledné priority pre jednotlivé kombinácie ich hodnôt. Nedozvieme sa ani na základe akých konkrétnych kritérií máme vyhodnocovať, či incident má práve tú a tú hodnotu dopadu a potreby pre používateľa.[24]

Podľa ITIL nie je možné systém riadenia služieb IT objektívne auditovať

Pretože ITIL skoro nič neprikazuje, ale skoro všetko iba odporúča, nedá sa u konkrétneho systému riadenia služieb v konkrétnej organizácii objektívne prehlásiť, že ten a ten jeho prvok je implementovaný zle alebo dobre.[24]

ITIL neobsahuje prevratné novinky

ITIL sa často spomína nielen ako best practice, ale zároveň aj ako common sense (zdravý rozum). Aj v organizáciách, kde o ITIL nikdy nikto nepočul, môže existovať systém riadenia služieb, ktorý obsahuje mnoho prvkov best practice popísaných v ITIL publikáciách, a to jednoducho preto, že IT manažéri a špecialisti v tejto organizácii dospeli k rovnakým znalostiam, ako mnohí iní pred nimi.[24]

3.1.1 Incident management

Kľúčové pojmy:

- Incident = neplánované prerušenie služby IT alebo zníženie jej kvality . Incidentom je taktiež porucha konfiguračnej položky, ktorá zatiaľ nemala dopad na funkčnosť služieb IT.
- Náhradné riešenie (workaround) = obmedzenie, alebo vylúčenie dopadu incidentu, alebo problému, pre ktoré zatiaľ nie je k dispozícii úplné riešenie.

Hlavné ciele a poslanie procesu:

- Obnoviť normálnu prevádzku služby, a to čo najrýchlejšie pri súčasnej minimalizácii dôsledkov výpadku služby na prevádzku (tzn. na zákazníkov a používateľov).

Stručný popis procesu a jeho hlavné charakteristiky:
Proces je zodpovedný za včasnú detekciu incidentov, ich zapísanie, a riadenie ich životného cyklu. Jeho cieľom je byť čo najrýchlejší. Incident management v zásade neskúma, prečo k incidentom dochádza (za hľadanie príčin incidentov je zodpovedný proces problem management), ale hľadá akékoľvek riešenie vrátane workaroundu, ktorého výsledkom bude obnovenie služby a/alebo eliminácia, či aspoň obmedzenie dôsledkov výpadku služby. Incident management teda nemôže ovplyvniť počet incidentov, ale iba dĺžku ich trvania a obchodné straty vyplývajúce z ich existencie. Kľúčovým znakom procesu je riešenie incidentov v poradí podľa priority stanovenej na základe biznis dôležitosti ohrozených služieb IT. Incidenty sú najčastejšie detekované procesom event management (riadenie udalostí) a používateľom, ktorý kontaktuje service desk.

Proces incident management by mal byť čo najviac automatizovaný pomocou vhodného nástroja, vrátane rýchleho prístupu k informáciám o predchádzajúcich incidentoch a ich riešeniach (znalostná databáza), ako aj k informáciám o architektúre služieb IT a súvisiacej technickej infraštruktúre.[25]

Najdôležitejšie prínosy procesu:

- Zmenšenie dôsledkov dopadov incidentu
- Skrátenie dĺžky trvania výpadkov služieb IT
- Zvýšenie spokojnosti zákazníkov a používateľov

3.1.2 Problem management

Kľúčové pojmy:

- Problém = príčina jedného alebo viacerých incidentov
- Známa chyba (known error) = problém, ktorý má zdokumentovanú hlavnú príčinu a náhradné riešenie

Hlavné ciele a poslanie procesu:

- Zodpovedá za riadenie všetkých problémov počas celej doby ich životného cyklu
- Proaktívne znižovať výskyt incidentov
- Riadiť znalostnú databázu a informácie v nej uložené sprístupňovať špecialistom podpory v procese incident management a operátorom service desku

Stručný popis procesu a jeho hlavné charakteristiky:
Proces zodpovedá za včasnú detekciu problémov, ich záznamy a riadenie ich životného cyklu. Jeho cieľom je zisťovať príčiny incidentov a spôsoby ich odstránenia, rozhodovať o trvalom odstránení chyby, alebo o jej dočasnom, či trvalom zachovaní v IT infraštruktúre, smerovať do procesu change management žiadosti o zmenu, ktorých obsahom je odstránenie chyby z infraštruktúry, a potom, keď je chyba skutočne odstránená, overiť, či pôvodné symptómy boli odstránené a k incidentom spôsobených touto chybou už nedochádza.

Proces taktiež riadi znalostnú databázu, do ktorej zapisuje identifikované známe chyby a postupy (reakcie), ktorými proces reagoval na chybami vyvolané incidenty; taktiež sem zapisuje náhradné riešenia (workarounds) k incidentom, u ktorých sa ešte nepodarilo identifikovať ich príčinu.[26]

Najdôležitejšie prínosy procesu:

- Zvýšenie dostupnosti služieb IT
- Zníženie výskytu incidentov, zaistenie stability infraštruktúry
- Zvýšenie úspešnosti service desku v kritériu „first-time fix“ (ako dôsledok naplnenia znalostnej databázy)

3.1.3 Service Desk

Kľúčové pojmy:

- Jediné kontaktné miesto (single point of contact) = poskytnutie jediného konzistentného spôsobu komunikácie s organizáciou alebo podnikovou jednotkou.
- Typický service desk spravuje incidenty a požiadavky na službu a obstaráva komunikáciu s používateľmi.

Základné povinnosti, ktoré musí service desk zaistiť:

- Fungovať ako jediné kontaktné miesto pre používateľa služieb.
- Obnoviť normálny beh služby v prípade jej výpadku, a to čo najrýchlejšie = zastáva rolu 1. úrovne podpory v procesu incident management.
- Spracovávať všetky požiadavky od používateľov služieb a zaisťovať ich vybavenie = odpovedá za celý proces request fulfilment.

Táto práca sa zaoberá čiastočným spracovaním Service Desku, ktorý vypúšťa proces request fulfilment. Stará sa len o incident management a problem management, spolu s ďalšou funkcionalitou. Jedná sa teda o zjednodušený Service Desk blížiaci sa k Help Desku.[27]

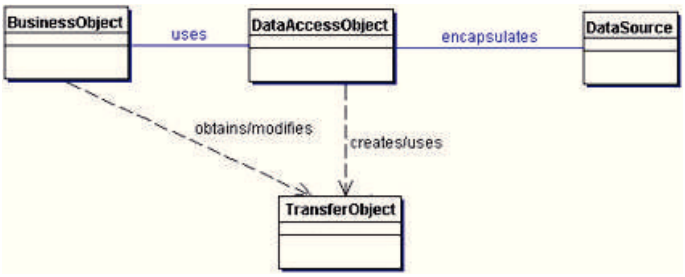
3.2 Návrhové vzory

Keďže táto práca sa nezaoberá problematikou a opisom návrhových vzorov, popíšem len vzor `DAO`, spolu s jeho variáciami `DAO Factory` a `Abstract DAO Factory`, ktorý sa priamo týka požiadavky zadania : *Implementované komponenty budou schopné poskytovat rozhraní pro použití jakékoliv databázové vrstvy*.

3.2.1 Data access object - DAO

Špecifikácia pre J2EE:

Tento návrhový vzor používame na zabstraktnenie a zapuzdrenie prístupu k dátovému zdroju. `DAO` sa stará o pripojenie k dátovému zdroju z ktorého získava alebo naň ukladá dáta a implementuje prístupový mechanizmus, ktorý je potrebný na prácu s týmto zdrojom. Biznis komponent, ktorý používa `DAO` využíva jednoduché rozhranie, ktoré `DAO` poskytuje svojim klientom. `DAO` teda kompletne skrýva implementáciu dátového zdroja pred svojim klientom. Pretože rozhranie, ktoré poskytuje klientom sa nemení v prípade, že sa zmení implementácia dátového zdroja, tak tento fakt dovoľuje `DAO` prispôsobiť sa rôznym schémam ukladania dát bez toho, aby ovplyvnilo svojich klientov alebo biznis komponenty. V podstate sa `DAO` správa ako adaptér medzi komponentom a dátovým zdrojom. Obr. 6 zobrazuje triedny diagram `DAO Factory`.



Obr. 6: Triedny diagram DAO Factory

Súčasti DAO:

- **BusinessObject** - Biznis objekt Reprezentuje dátového klienta. Je to objekt, ktorý vyžaduje prístup k dátovému zdroju, či už z dôvodu uloženia alebo získavania dát.
- **DataAccessObject** - Objekt prístupu k dátam Je hlavným objektom tohto návrhového vzoru. Tento objekt zabstraktňuje implementáciu dátového zdroja pred biznis objektom, aby tak umožnil transparentný prístup k dátovému zdroju. Biznis objekt taktiež deleguje operácie načítania a ukladania dát na DataAccessObject.
- **DataSource** - Dátový zdroj Reprezentuje implementáciu dátového zdroja. Dátový zdroj môže byť RDBMS, OODBMS, XML repository a iné.
- **TransferObject** - Reprezentuje objekt, ktorý slúži na prenášanie dát. DataAccessObject môže používať TransferObject na vrátenie dát klientovi, ale taktiež na prijímanie dát od klienta, ktoré majú byť následne aktualizované v databáze.

Obr. 16 zobrazuje sekvenčný diagram popisujúci `DAO Factory`.

Ak vieme, že sa dátový zdroj bude meniť (vyplýva to zo zadania práce), môže byť táto stratégia implementovaná za pomoci návrhového vzoru `Abstract Factory`. `Abstract Factory` môže naopak stavať na a používať návrhový vzor `Factory`. V tomto prípade táto stratégia poskytuje abstraktný `DAO factory` objekt (`Abstract Factory`), ktorý môže vytvárať rôzne typy `DAO factories`, pričom každá `factory` podporuje odlišný typ dátového zdroja. Potom, čo získame `DAO factory` pre špecifický zdroj, použijeme ju na získanie konkrétneho `DAO` objektu určeného pre tento zdroj. Táto stratégia je zobrazená na obr. 17.

Diagram zobrazuje základnú `Abstract DAO factory`, čo je abstraktná trieda, ktorá je dedená a implementovaná konkrétnymi `DAO továrňami`, aby bol zabepečený špecifický prístup na základe použitého dátového zdroja. Klient môže získať konkrétnu implementáciu `DAO factory` (ako na obr. 17 RdbDAOFactory) a použiť ju, aby získal konkrétne `DAO` triedy, ktoré pracujú s týmto dátovým zdrojom. Jednotlivé `DAO` triedy môžu extendovať resp. implementovať generickú základnú triedu (na obr. je to DAO1 a DAO2), ktorá špecifikuje, aké metódy má `DAO` obsahovať na základe biznis objektu pre ktorý je určené. Základný princíp fungovania `Abstract DAO Factory` je ukázaný na obr. 18 [28]

Poznámka 3.1 V Java EE 6 (vlastne už od verzie 5) sa veci zmenili s príchodom JPA - Java Persistence API. JPA používa EntityManager, ktorý v podstate zastáva funkciu DAO. [4]

Keďže ja v práci používam vlastné ORM za pomoci čistého JDBC, DAO zohráva stále dôležitú úlohu v otázke prenositeľnosti, ako už bolo spomenuté.

4 Špecifikácia funkčnosti

V tejto sekcii upresním požiadavky zadania.

4.1 Upresnenie zadania

Základom celého IS má byť modul používateľov, skupín, rolí a práv, ku ktorým má byť pridaný incident management modul. K týmto modulom som doplnil naviac aj modul na správu vybavenia, a taktiež problem management modul a vytvoril tak plnohodnotnejšiu aplikáciu.

Poznámka 4.1 Modul práv je úzko spätý s modulom rolí, spolu vytvárajú jeden komplexnejší modul.

Jednotlivé moduly sa budú skladať z operácií, ako je **vytvorenie** novej jednotky modulu a jej **správa**. Pod správou jednotky si môžeme predstaviť ďalšie základné operácie, ako jej editácia, vymazávanie, zobrazenie a pod. K incident a problem management modulu bude tiež, okrem iného, pridaná možnosť priradzovať incidenty a problémy technikovi, a taktiež možnosť uložiť ich do znalostnej databázy, teda veľmi zjednodušene, do histórie incidentov, či problémov. Ďalej musí byť umožnené naviazať incident s určitým problémom, čo logicky vyplýva z opisu procesu problem management viz. 3.1.2, atď. Konkrétny zoznam operácií(akcií) pre niektoré moduly :

- **Modul používateľov**
  - Vytvorenie nového používateľa
  - Zobrazenie detailov používateľa
  - Zmena skupiny používateľa
  - Zmena roly používateľa
  - Zmazanie používateľa
  - Povýšenie používateľa na super admina
- **Modul skupín**
  - Vytvorenie novej skupiny
  - Zobrazenie detailov skupiny
  - Editácia skupiny
  - Zmazanie skupiny

- **Modul rolí**
  - Vytvorenie novej roly
  - Zobrazenie detailov roly
  - Editácia roly - zmena oprávnení
  - Zmazanie roly

Kompletný prehľad je dostupný na priloženom CD.

4.1.1 Používateľské roly

Ako je spomínané v časti 3.1, procesy problém a incident management majú byť spravované rôznymi osobami. Podobne ostatné moduly budú riadené rôznymi osobami, resp. pre každý proces bude vyhradená používateľská rola. Taktiež bude dostupná rola super admina[3], ktorý má prístup ku všetkým modulom a akciám. IS budú používať samozrejme aj bežní užívatelia (vytvorení správcom používateľov), ktorí nebudú zodpovední za správu žiadneho procesu, budú mať len základné práva, ako je vytvorenie incidentu. V neposlednom rade budú s IS pracovať aj technici zodpovední za riešenie problémov a incidentov. Prehľadný zoznam základných rolí je zobrazený v tab. 1. Systém umožní tiež definovať vlastné roly podľa potreby spoločnosti a priradiť im dostupné akcie.

Alias	Rola	Akcie
Common user	Bežný používateľ	Vytváranie incidentov,...
Super admin	Správca so všetkými právami	Všetky práva
Incident admin	Správca incidentov	Editácia incidentu,...
Problem admin	Správca problémov	Vytváranie problému,...
User admin	Správca používateľov	Vytváranie používateľa,...
Group admin	Správca skupín	Editácia skupín, vytváranie,...
Equipment admin	Správca zariadení	Pridávanie do DB, editácia,...
Technician	Technik	Riešenie problémov, incidentov,..

Tabuľka 1: Tabuľka používateľských rolí

[3]Bežne sa výraz super admin chápe ako správca celého systému. V tomto prípade sa nejedná o správcu systému ako takého. Je to len používateľ so všetkými právami vzhľadom na možnosti IS.

4.1.2 Akcie rolí

Každý používateľ má právo :

- Vytvárať incidenty
- Zobraziť si prehľad svojich aktuálne vytvorených incidentov
- Editovať svoje incidenty
- Zmazať svoje incidenty
- Editovať profil a meniť svoje heslo

Ukážkový prehľad akcií (oprávnení) k niektorým základným roliam :

- **Technik**
  - Zobrazenie svojich priradených incidentov
  - Zobrazenie histórie incidentov
  - Pridanie workaroundu pre incident
  - Zobrazenie svojich priradených problémov
  - Zobrazenie histórie problémov
  - Pridanie riešenia pre problém
- **Správca používateľov**
  - Vytvorenie nového používateľa
  - Zmena roly používateľa
  - Zmena skupiny používateľa
  - Zmazanie používateľa

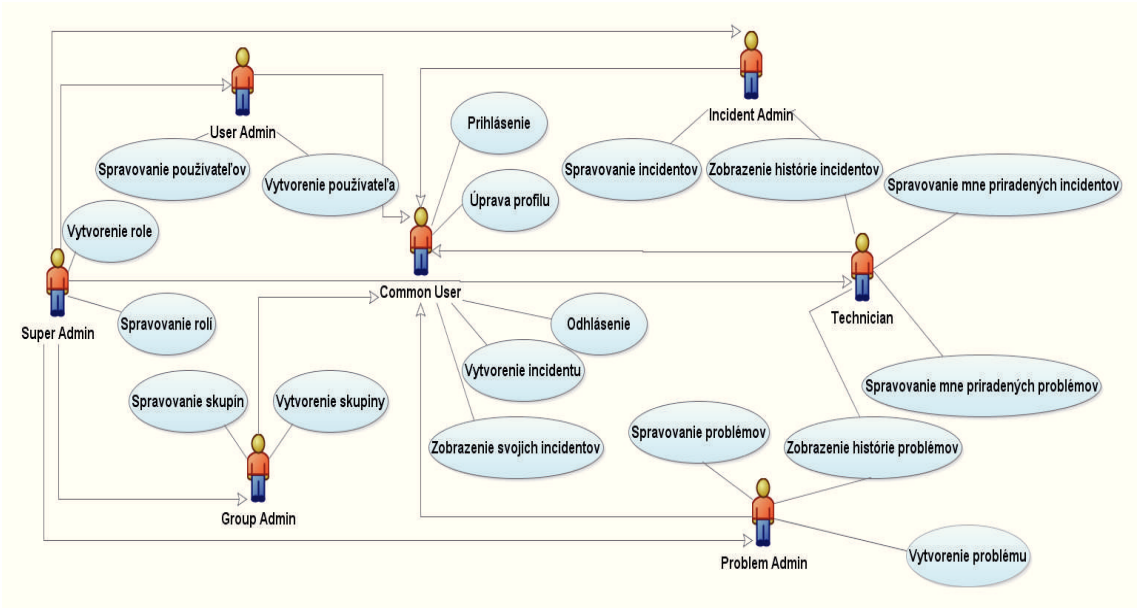
  Kompletný prehľad je dostupný na priloženom CD.

4.1.3 Grafické vyjadrenie požiadaviek

Z dôvodu veľkosti uvediem len Use case diagram úrovne 1.

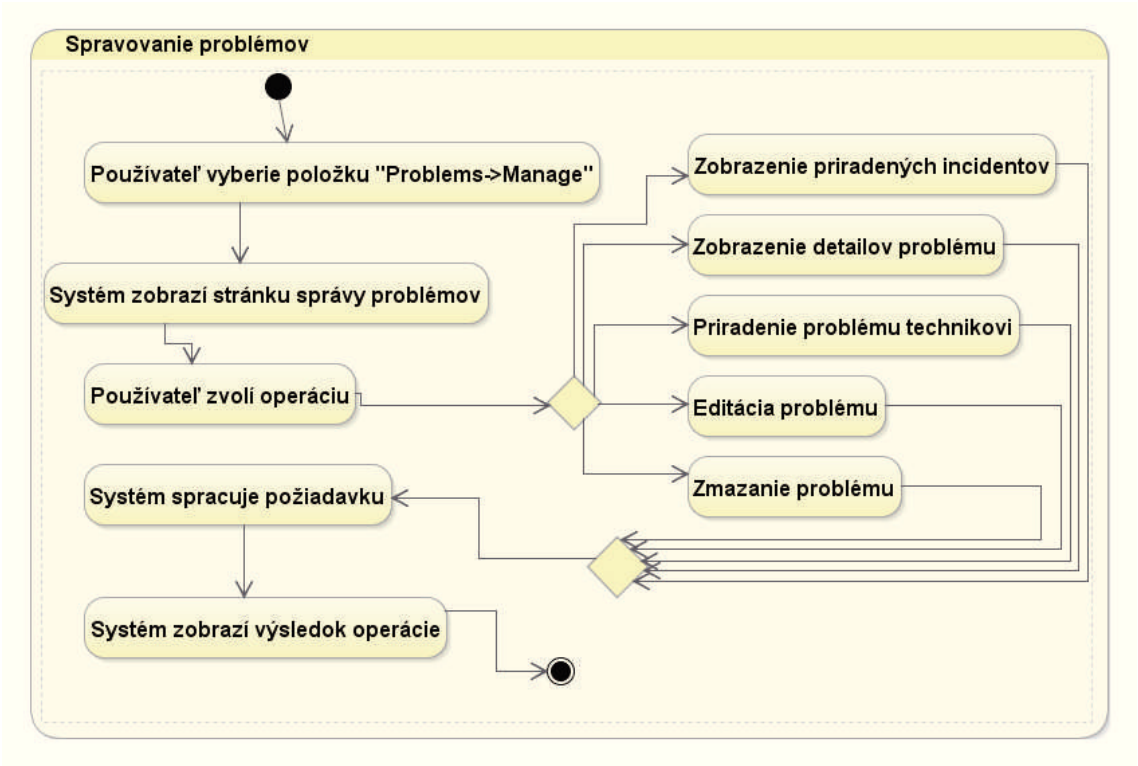
Funkčnosť IS teda môžeme zhrnúť nasledovne :

IS bude pre používateľov dostupný cez webový prehliadač. Budú ho využívať viaceré pobočky firmy. Užívatelia IS majú priradené svoje roly a patria do určitej skupiny. Rola definuje práva používateľov, zatiaľ čo skupina obmedzuje viditeľnosť dát, čiže napr. správca používateľov pobočky v Ostrave môže spravovať len používateľov z tejto pobočky (skupiny). Bežní používateľ má právo, okrem iného, vytvárať incidenty. Vzniknuté incidenty správca incidentov priradí technikovi, aby bolo čo najrýchlejšie obnovené fungovanie vyradenej služby. Neskôr je skúmaná podstata incidentov, správca problémov vytvorí problém, priradí mu dané incidenty a technika, ktorý ho bude riešiť.



Obr. 7: Use case diagram, úroveň 1: Roly a ich oprávnenia

Po vyriešení sa problémy a incidenty presúvajú do histórie, ktorá tvorí zároveň zjednodušenú znalostnú databázu. V hierarchii používateľov, čo sa týka práv, je super-admin na vrchole. Ten môže vykonávať všetky funkcie ostatných používateľov a navyše definovať nové roly. Ostatné roly systému, ako už bolo spomenuté, sú zobrazené v tab. 1.



Obr. 8: Activity diagram: Spravovanie problémov

5 Analýza a návrh

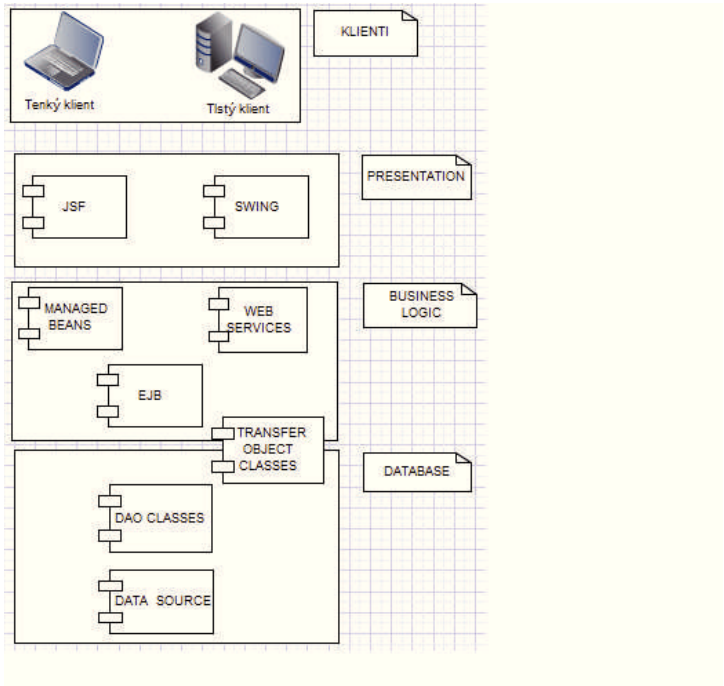
V tejto sekcii uvediem návrh štruktúry aplikácie, jej modulov, jednotlivých vrstiev, tried a tabuliek. Z dôvodu rozsiahlosti aplikácie uvediem len ukážky jednotlivých diagramov.

5.1 Návrh štruktúry aplikácie

Keďže sa jedná o webovú Java EE aplikáciu, rozhodol som sa ju riešit' klasicky, pomocou 3-vrstvej architektúry, tzn. tvoria ju nasledujúce vrstvy:

- **Prezentačná vrstva (Presentation)** - Zobrazuje informácie pre používateľa, môže kontrolovat' zadávané vstupy, neobsahuje však spracovanie dát.
- **Aplikačná vrstva (Business logic)** - Je jadrom aplikácie, prebieha tu spracovanie dát.
- **Dátová vrstva (Database)** - Túto vrstvu tvorí najčastejšie databáza, ktorá dáta uchováva, sprístupňuje a zaručuje ich konzistenciu.

V našom prípade budú jednotlivé vrstvy obsahovat' komponenty, ako je uvedené na obr. 9:



Obr. 9: Model komponent

Poznámka 5.1 V diagrame komponentov som znázornil aj možné budúce rozšírenie o tlstého klienta - Java Swing rozhranie a biznis logika by bola volaná pomocou webových služieb.

Ukážkový tok dát cez komponenty od prezentačnej vrstvy po dátovú je teda : `JSF` → `Managed Bean` → `EJB` → `DAO` → DB

5.2 Návrh všeobecného riešenia

Je samozrejmé, že IS bude prístupný len registrovaným používateľom. Nutnosťou je teda implementovať login. Z požiadaviek vyplýva, že registráciu nového používateľa bude mať pod kontrolou User admin. Tým bude zabezpečená väčšia réžia nad používateľmi. Musíme teda zamedziť aj prihlasovaniu viacerých používateľov z jedného účtu v rovnakom čase. Keďže k systému budú pristupovať používatelia s rôznymi rolami musí byť zaručené, že budú mať prístup len k funkciám, na ktoré im ich rola dáva oprávnenie. Treba teda obmedziť viditeľnosť komponentov na stránke, a taktiež kontrolovať právo na prístup k danej URL adrese, pretože síce neposkytneme používateľovi možnosť dostať sa k stránke pomocou tlačidla, ale stále ju môže napísať priamo do vyhľadávača. Podľa popisu rolí budú vytvorené základné roly systému. Pre jednotlivé akcie (práva) bude vytvorená DB tabuľka a rovnako tak aj pre prístupové práva k jednotlivým stránkam.

5.3 Návrh databáze

Lineárny zápis entít (**primárny kľúč**,*cudzí kľúč*):

`User`(user_name, **user_id**, user_password, first_name, last_name, admin, email, building, office, *role_id*,*group_id*)

`Group`(**group_id**, group_name, group_desc)

`Actions`(**action_id**, action_name, action_description)

`Actions2role`(**role_id**, **action_id**)

`Actions2Url`(**action_id**, **url_part**)

`Category`(**category_id**, category_name, category_description)

`Equipment`(**equip_id**, equip_desc, equip_model, equip_manufact, equip_serial_num, equip_building, equip_office, *equip_type_id*)

`Equipment_types`(**equip_type_id**, equip_type_name, equip_type_desc)

`Incident`(**incident_id**, inc_title, inc_description, inc_solution, inc_created, inc_solved, inc_workaround, *priority_id*, *status_id*, *problem_id*, *category_id*, *user_id*, *equip_id*)

`Incident_history`(**incident_id**, inc_title, inc_description, inc_solution, *priority_id*, inc_created, inc_solved, *status_id*, *problem_id*, *category_id*, *user_id*, *equip_id*)

`Priority`(**priority_id**, priority_name)

`Problem`(**problem_id**, prob_title, prob_description, prob_solution, prob_created, prob_solved, *priority_id*, *status_id*, *category_id*, *user_id*, *equip_id*)

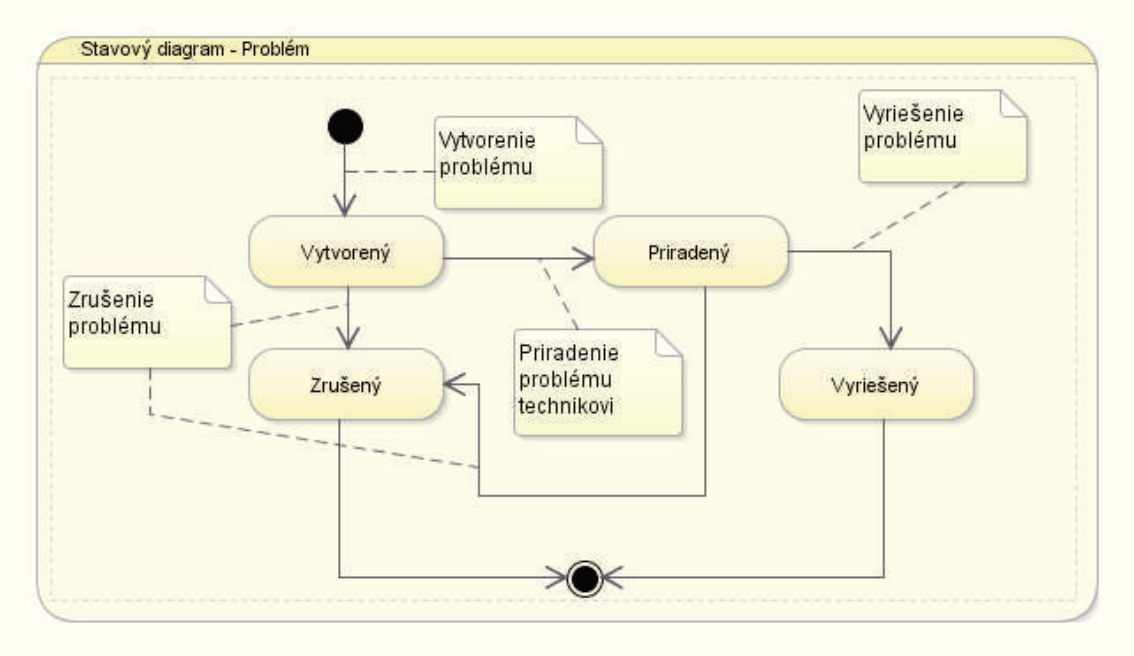
`Problem_history`(**problem_id**, prob_title, prob_description, prob_solution, prob_created, prob_solved, *priority_id*, *status_id*, *category_id*, *user_id*, *equip_id*)

`Role`(**role_id**, role_name, role_desc)

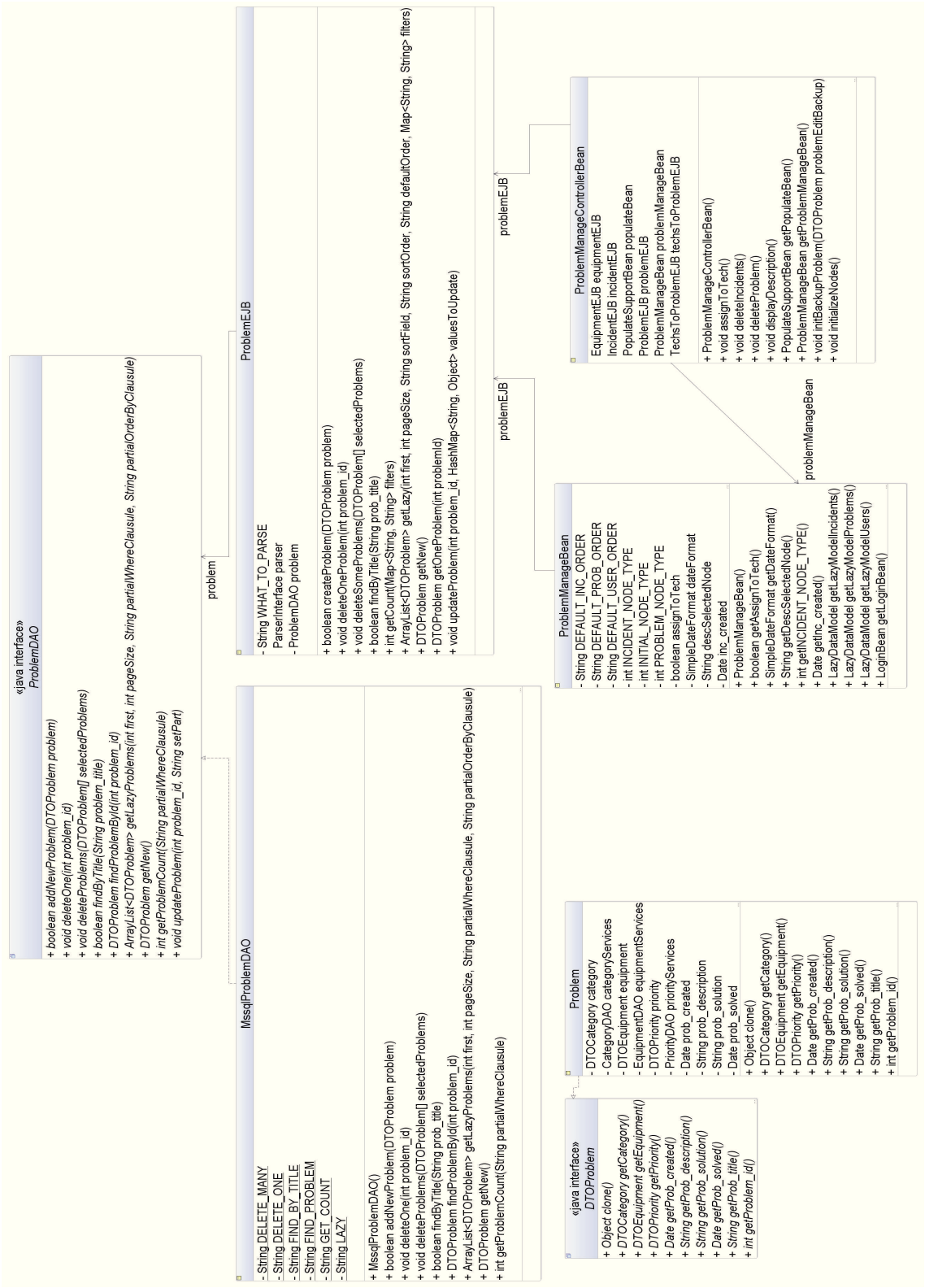
`Status`(**status_id**, status_name)
`Sub_category`(**sub_category_id**, sub_category_name, sub_category_desc, *category_id*)
`Techs2incident`(**incident_id**, **user_id**)
`Techs2problem`(**problem_id**, **user_id**)
Tabuľky, ktoré vyplývajú zo zadania a návrhu riešenia sú zobrazené na obrázku 19.

5.4 Návrh modulov

Pre ukážku uvediem len návrh Problem management modulu. Operácie pre tento modul sú popísané v prehľade jednotlivých modulov, ktorý sa nachádza v prílohe na CD. Triedne diagramy sú príliš veľké na to, aby som ich umiestnil do textu práce, preto uvádzam len ich zjednodušené podoby. Detailnejšie sú spracované v prílohe na CD.



Obr. 10: Stavový diagram: problém



Obr. 11: Java class diagram: Správa problémov - zjednodušený

6 Implementácia

V tejto sekcii popíšem implementačné detaily najzaujímavejších častí kódu. Kompletné zdrojové kódy sú dostupné v prílohe.

Ukážkový systém obsahuje cez 150 tried a konfiguračných súborov, zdrojový kód presahuje 14 500 riadkov. Databáza obsahuje 17 tabuliek, 21 procedúr a 8 triggerov.

Do tejto sekcie som sa snažil zaradit' len najdôležitejšie časti implementácie, keďže jej rozsah mi zďaleka neumožňuje pokryt'všetko. Ukážková funkčnost'aplikácie je zachytená v krátkom prezentačnom videu na priloženom CD.

Konfigurácia:

- Java EE 6 + JDK 1.7
- JSF 2.0
- Glassfish 3.1
- Primefaces 3.1.1
- iText-2.1.7
- redmond 1.0.3
- JDBC driver: sqljdbc4
- Microsoft SQL Server 2008 R2

6.1 Primefaces komponenty

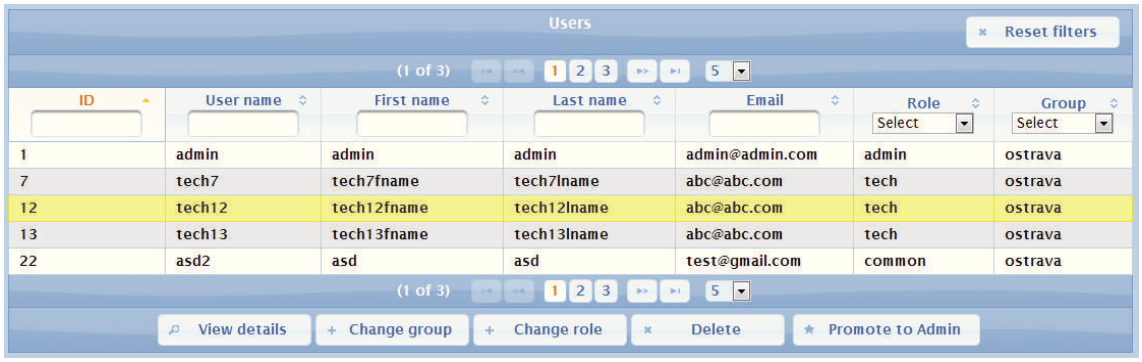
<p:panelGrid>	<p:inputText>	<p:ajax>
<p:column>	<p:calendar>	<p:dataList>
<p:commandButton>	<p:outputPanel>	<p:separator>
<p:dataTable>	<p:dataExporter>	<p:tooltip>
<p:growl>	<p:confirmDialog>	<p:tree>
<p:contextMenu>	<p:password>	<p:picklist>

Tabuľka 2: Tabuľka použitých komponentov

Popis všetkých komponentov je dostupný v používateľskej príručke na priloženom CD. Ďalej uvediem len stručný popis najpodstatnejších komponentov a ich použitie v mojej práci.

6.1.1 DataTable

`Primeface DataTable` predstavuje vylepšenú verziu klasickej `datatable`. Poskytuje integrované riešenia pre rôzne klasické požiadavky, ako je stránkovanie, triedenie, výber, „lenivé“ načítavanie , filtrovanie a iné.



Obr. 12: Ukážka dataTable z aplikácie

Stránkovanie

`DataTable` má vstavaný AJAX „stránkovač“ (paginator), ktorý nastavujeme pomocou atribútu paginator.

Triedenie

Tým, že definujeme u stĺpca tabuľky atribút sortBy , povolíme naň AJAX triedenie. Defaultne sa používa triediaci algoritmus, ktorý poskytuje Java Comparator, ale `DataTable` poskytuje možnosť použiť aj vlastnú metódu definovaním sortFunction. `DataTable` môže zobraziť dáta, ktoré sú defaultne zotriedené, stačí u nej definovať atribút sortBy. Môžeme taktiež definovať smer triedenia (zostupne/vzostupne) pomocou atribútu sortOrder.

Filtrovanie

Podobne ako triedenie môžeme na stĺpec nastaviť aj AJAX filtrovanie definovaním atribútu filterBy . Filtrovanie je vyvolané po udalosti keyup. Vstupy u filtrov môžu byť naštýlované pomocou atribútov filterStyle a filterStyleClass . Namiesto čistého textového vstupu je tiež možné použiť rozbaľovaciu ponuku, a to tak, že nastavíme atribút filterOptions a kolekciu alebo pole selectItemov. Ďalej je tiež možné nastaviť atribút filterMatchMode, ktorý určuje akým spôsobom sa má určovať zhoda dát s filtrom (startsWith, exact, ...).

Výber riadku

Existuje niekoľko spôsobov, ako môžeme vybrať riadok z tabuľky. Najzákladnejší je pomocou p:commandButton alebo p:commandLink a použitia <f:setPropertyActionListener>.

```
<p:column>
    <p:commandButton value="Vyber">
        <f:setPropertyActionListener value="#{fooObject}" target="#{fooBean.selectedFooObject}"/>
    </p:commandButton>
</p:column>
```

Výpis 4: Výber riadku DataTable pomocou <f:setPropertyActionListener>

Ak chceme vybrať riadok pomocou kliknutia na riadok, namiesto p:commandButton stačí definovať atribút selectionMode s hodnotami single alebo multiple podľa toho, či chceme povoliť výber jednoho alebo viacerých riadkov naraz. V tomto prípade prebehne výber riadku po jednom kliknutí. Ak by sme chceli vybrať riadok po dvojkliku tak stačí

použit' atribút dblClickSelect. Často býva bežnou požiadavkou, aby mala `DataTable` jeden stĺpec v ktorom by bol `radioButton` umožňujúci výber daného riadku. Na túto požiadavku team `Primefaces` tiež myslel, stačí len definovat' atribút selectionMode s hodnotou single u daného stĺpca. Podobne môžeme povolit' aj výber viacerých riadkov za použitia `checkboxov`, stačí opät' definovat' atribút selectionMode, ale teraz s hodnotou multiple.

Lenivé načítavanie (Lazy loading)

Táto vstavaná funkcia zefektívňuje prácu s rozsiahlymi zdrojmi dát. Normálny AJAX paginator síce vykresľuje len danú stránku, ale aj tak vždy načíta všetky dáta (nie len z aktuálne vykreslenej strany). `DataTable`, ktorá podporuje lazy loading pracuje podobne, čo sa týka vykreslenia, avšak vždy načíta len potrebné údaje pre vykreslenie a nie všetky dáta. Aby sme mohli implementovat' lazy loading potrebujeme ako hodnotu tabuľky value zvolit' org.primefaces.model.LazyDataModel a implementovat' jeho load metódu. Taktiež musíme implementovat' metódy getRowData a getRowKey, ak je povolený výber riadkov. `DataTable` volá load metódu po každom stránkovaní, triedení, či filtrovaní. Load metóda má nasledujúce parametre:

- first : Indikátor, od ktorého záznamu začat' načítavat' dáta
- pageSize: Počet dát určených na načítanie
- sortField: Názov atribútu podľa ktorého sa triedi (napr. `fooColor` pre sortBy = "{foo.fooColor}")
- sortOrder: SortOrder enum. Obsahuje atribút, podľa ktorého sa budú záznamy triedit' a hodnotu triedenia : zostupne/vzostupne
- filters : Map<String, String> s názvom atribútu ako kľúčom (napr.`fooColor` pre filterBy = "{foo.fooColor}") a hodnotou atribútu ako hodnotou.

Okrem load metódy treba poskytnút' aj totalRowCount (celkový počet riadkov), aby paginator zobrazil správny počet strán.

Bližší popis ďalších funkcií `DataTable` je v používateľskej príručke v prílohe.

Tento komponent využívam v práci najčastejšie, pretože vďaka možnosti zoraďovania, filtrovania, stránkovania a podpore lenivého načítavania sa jedná o najefektívnejší spôsob zobrazovania rozsiahlych dát. V kombinácii s komponentmi, ako `Dialog`, `Tree`, či `ContextMenu` vytvára naozaj flexibilné GUI.

Implementácia

Popíšem `DataTable`, ktorú využívam na stránke správy incidentov, ostatné sú jej veľmi podobné.

Kód `faceletu` som pre väčšiu prehľadnost' rozdelil do 4 častí s popisom:

Nastavenia komponentu

Definujem tu základné nastavenia, ktoré som spomínal, mimo iné aj paginatorTemplate podľa ktorej sa vykreslí paginátor.

```
<p:dataTable id="tblIncMain" var="dataItem" value="#{incManageBean.lazyModelIncidents}"
    rows="#{incManageBean.pageSize}" paginator="true" paginatorTemplate="{
    CurrentPageReport} {FirstPageLink} {PreviousPageLink} {PageLinks} {NextPageLink}
    {LastPageLink} {RowsPerPageDropdown}" rowsPerPageTemplate="5,10,15"
    resizableColumns="true" widgetVar="tblIncMainWidget" selection="#{incManageBean.
    selectedIncidents}" selectionMode="multiple">
```

Výpis 5: DataTable facelet - nastavenia

Hlavička

Hlavičku `DataTable` špecifikujeme pomocou značky <f:facet name="header">. V tomto prípade obsahuje hlavička komponent `Tooltip`, ktorý umožňuje zobraziť nápovedu pre komponent, na ktorý ho naviažeme. Ďalej sa tu nachádza tlačidlo, pomocou ktorého resetujem filtre tabuľky.

Poznámka 6.1 Ak aktualizujeme hodnoty nejakého komponentu treba ho po aktualizácii updatovať, aby boli obnovené nové hodnoty v jeho modelu.

Taktiež tu používam `commandLink`, ku ktorému je priradený komponent `DataExporter`, ktorý umožňuje exportovanie dát `DataTable` do PDF pomocou knižnice `iText`.

```
<f:facet name="header">
  <p:outputPanel>
    <h:outputText id="incManageTooltip" value="Unsolved Incidents" styleClass="tableHeaders"/
        >
    <p:tooltip for="incManageTooltip" styleClass="tooltips" value="Solved incidents can be
        found in Incidents history." showEffect="fade" hideEffect="fade" />
    <p:commandButton value="Reset filters" style="width: 14\%; margin: 2px; float: right;" ajax
        ="true" icon="ui-icon ui-icon-close" update=":frmMainIncManage:tblIncMain" onclick="
        tblIncMainWidget.clearFilters();dateFilter.setDate(null)"/>
    <h:commandLink>
      h:outputText style=" float: left ;" value="Export to PDF" />
      <p:dataExporter type="pdf" target="tblIncMain" fileName="incidents" pageOnly="true"/>
    </h:commandLink>
  </p:outputPanel>
</f:facet>
```

Výpis 6: DataTable facelet - hlavička

Stĺpce

Názvy stĺpcov v `DataTable` nemusíme definovať pomocou <f:facet name = "header "> tak, ako tomu je v `JSF`. `Primefaces` na prácu so stĺpcami poskytujú komponent `Column`, kde názov definujeme v atribúte headerText. V ukážke sú zahrnuté rôzne typy filtrovania stĺpcov, funkcia ellipsis, ktorú spomeniem v sekcii 6.3 a taktiež ukážka použitia komponentu `Calendar`.

```
<p:column id="titleColumn" style="width: 150px" headerText="Title" sortBy="#{dataItem. inc_title }"
    filterBy ="#{dataItem. inc_title }">
  <h:outputText value="#{func:ellipsis(dataItem. inc_title , 15)}" />
</p:column>
<p:column style="width: 150px" sortBy="#{dataItem.inc_created}">
  <f:facet name="header">
    <h:outputText value="Created on" />
    <br />
    <p:calendar widgetVar="dateFilter" id="dateFilter " value="#{incManageBean.inc_created}"
        mode="popup">
      <!--metóda volaná v oncomplete vyvolá refresh filtrov tabuľky-->
      <p:ajax event="dateSelect" update=":frmMainIncManage:tblIncMain" oncomplete="
          tblIncMainWidget.filter()"/>
    </p:calendar>
  </f:facet>
  <h:outputText value="#{dataItem.inc_created}" />
</p:column>
<p:column style="width: 190px" headerText="Incident Category" sortBy="#{dataItem.category.
    category_name}" filterBy="#{dataItem.category.category_id}" filterMatchMode="exact"
    filterOptions="#{populateBean.categoryFilter}">
  <h:outputText value="#{dataItem.category.category_name}" />
</p:column>
```

Výpis 7: DataTable facelet - ukážka stĺpcov

Päta

Pätu definujeme pomocou <f:facet name="footer">. Zvyčajne sa tu definujú tlačidlá pre prácu s `DataTable`.

```
<f:facet name="footer">
  <p:commandButton value="View" style="width: 14\%; margin: 2px;" icon="ui-icon ui-icon-
      search" update=":frmMainIncManage:diaIncDetail" oncomplete="incidentDialog.show()"/>
</f:facet>
```

Výpis 8: DataTable facelet - päta

Najdôležitejšia je implementácia LazyDataModel. Jeho inicializácia prebieha v metóde volanej v @PostConstruct.

Poznámka 6.2 Rozdiel medzi @PostConstruct a obyčajným konštruktorom bez parametrov je taký, že v @PostConstruct máme dostupné `EJB` metódy a môžeme robiť `JNDI lookup`, zatiaľ čo v konštruktore to možné nie je.

Metóda getRowKey vracia jednoznačný identifikátor riadku a metóda getRowData vracia objekt podľa tohto identifikátora.

Ukážka implementácie LazyDataModelu :

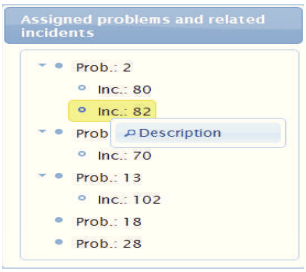
```
@PostConstruct
    private void init () {
        selectedIncidentForEdit = populateBean.getDummyIncident();
        dateFormat = new SimpleDateFormat("yyyy/MM/dd");
        lazyModelIncidents = new LazyDataModel<DTOIncident>() {

        @Override
        public Object getRowKey(DTOIncident object) {
            return object. getIncident_id () ;
        }

        @Override
        public DTOIncident getRowData(String rowKey) {
            return incidentEJB.getOneIncident(Integer.valueOf(rowKey));
        }

        @Override
        public List<DTOIncident> load(int first, int pageSize, String sortField , SortOrder
              sortOrder, Map<String, String> filters ) {
            if (inc_created != null) {
                filters .put("cast( floor (cast(inc_created as float ) ) as datetime)", dateFormat.format(
                    inc_created) + EXACT_MATCH);
            }
             filters .put("Group.group_id", String . valueOf(loginBean.getUser().getGroup().getGroup_id
                 ()) + EXACT_MATCH);
            lazyModelIncidents.setRowCount(incidentEJB.getCount(filters));
            return incidentEJB.getLazy( first , pageSize, sortField, sortOrder.toString () ,
                DEFAULT_INC_ORDER, filters);
        }
    };
}
```

Výpis 9: DataTable - LazyDataModel

Keďže nikdy neviem akú kombináciu filtrov používateľ využije musím `WHERE` časť `SQL` dotazu generovať dynamicky pomocou StringBuilder. Potom túto vybudovanú `WHERE` časť posielam priamo ako parameter do databázovej procedúry, ktorá mi vracia lazy dáta. V procedúre je potom dotaz vykonaný pomocou dynamického `SQL`. Vzniká teda riziko **SQL Injection**, viz. sekcia 6.2.2

6.1.2 Tree

`Tree` sa používa na zobrazenie hierarchicky usporiadaných dát a na vytváranie navigácie stránok. `Tree` je naplnený s inštanciami org.primefaces.model.TreeNode. Na to, aby sme mohli vytvoriť `Tree`, najskôr musíme definovať jeho koreň (root). Potom môžeme vytvoriť jednotlivé uzly stromu, ktoré priradíme koreňu, alebo už vytvorenému uzlu.



Obr. 13: Ukážka tree z aplikácie

```
TreeNode root = new DefaultTreeNode("Root", null);
TreeNode probNode = new DefaultTreeNode("problem", problem, root); //typ, dáta, rodič
TreeNode incNode = new DefaultTreeNode("incident", incident, probNode);

<p:tree id="fooTree" value="#{fooBean.root}" var="node">
    <p:treeNode type="problem" icon="ui-icon-bullet">
        <h:outputText value="#{node}" />
    </p:treeNode>
    <p:treeNode type="incident" icon="ui-icon-radio-off">
        <h:outputText value="#{node}" />
    </p:treeNode>
</p:tree>
```

Výpis 10: Ukážka vytvorenia Tree

`TreeNode API` sa teda používa na vytvorenie modelu, ktorý pozostáva z inštancií rg.primefaces.model.TreeNode a <p:treeNode> tag zasa reprezentuje komponent typu org.primefaces.component.tree.UITreeNode. TreeNode priradzujeme konkrétnemu <p:treeNode> pomocou názvu typu. Viď. ukážka 10.

Nedynamický Tree

Rozbaľovanie uzlov prebieha na strane klienta, takže na začiatku sa strom vykreslí a pošlú sa klientovi dáta o všetkých uzloch. Tento typ stromu je vhodný pre relatívne malý rozsah dát a poskytuje rýchlu interakciu s používateľom. Keďže sa klientovi pošlú všetky dáta (aj uzlov, ktoré nie sú rozbalené) tak tento typ nie je vhodný pre spracovanie rozsiahlych dát.

Dynamický Tree

Používa AJAX, aby načítaval uzly postupne zo strany serveru. V porovnaní s rozbaľovaním uzlov na strane klienta má dynamický strom výhodu pri práci s rozsiahlymi dátami, pretože sa pri inicializácii pošle klientovi len koreň a jeho deti. Inicializácia zvyšných uzlov prebieha pomocou lazy loadingu, takže pri rozbalení uzla sa pošlú len deti tohto uzla, nie celý strom.

Viacero typov TreeNode v jednom Tree

Je bežnou požiadavkou, aby mal strom viacero typov uzlov rozlíšených pomocou ikon. Túto požiadavku naplníme pomocou použitia viacerých typov uzlov a definovania ich ikon, viď. ukážka 10

Výber
Výber uzla je vstavaná funkcionalita stromu. Strom podporuje tri typy výberu :

- single: V rovnakom čase môže byt' vybraný len jeden uzol, priraďujeme do premennej typu TreeNode.
- multiple: V rovnakom čase môže byt' vybraných viacero uzlov, priraďujeme do premennej typu TreeNode[].
- checkbox: V rovnakom čase môže byt' vybraných viacero uzlov za pomoci checkboxov, priraďujeme do premennej typu TreeNode[].

ContextMenu
`Tree` má špeciálnu integráciu s `ContextMenu`. Je možné priradit' rôzne `ContextMenu` k rôznym typom uzlov pomocou atribútu nodeType u `ContextMenu`, v ktorom určíme typ uzla pre ktorý je `ContextMenu` určené.

Bližší popis `Tree` komponentu je v používateľskej príručke na priloženom CD.

V práci využívam `Tree` na zobrazenie aktuálne priradených problémov a ich incidentov pri zobrazení detailov technika v modulu problem management. `ContextMenu` som použil tak, aby bolo možné zobrazit' detaily incidentu či problému a odpojit' problém od technika.

```
<p:panel id="treeViewPanel" header="Assigned problems and related incidents">
  <p:tree id="treeTechView" value="#{probManageBean.root}" var="node" selectionMode="single"
      selection="#{probManageBean.selectedNode}">
    <p:treeNode type="problem" icon="ui-icon-bullet">
      <h:outputText value="#{node}" />
    </p:treeNode>
    <p:treeNode type="incident" icon="ui-icon-radio-off">
      <h:outputText value="#{node}" />
    </p:treeNode>
  </p:tree>

  <p:contextMenu for=":frmTechView:treeTechView" nodeType="problem">
    <p:menuitem value="Description" icon="ui-icon-search" ajax="true" actionListener="#{
        probManageControllerBean.displayDescription()}" update=":frmTechView:descTreePanel
        "/>
    <p:menuitem value="Unassign" icon="ui-icon-close" ajax="true" actionListener="#{
        probManageControllerBean.unassignProblemFromTech()}" update=":
        frmMainProbManage:tblProbMain,:frmTechView:treeTechView,:frmMainProbManage:
        tblTechs" />
  </p:contextMenu>

  <p:contextMenu for=":frmTechView:treeTechView" nodeType="incident">
    <p:menuitem value="Description" icon="ui-icon-search" ajax="true" actionListener="#{
        probManageControllerBean.displayDescription()}" update=":frmTechView:descTreePanel
        "/>
  </p:contextMenu>
</p:panel>
```

Výpis 11: Tree - facelet

Inicializácia tohto stromu potom prebieha pri otvorení dialógu, v ktorom sa nachádza. V podstate zistím problémy, ktoré sú priradené danému technikovi a vytvorím z nich uzly stromu typu problem, zároveň pre každý problém nájdem súvisiace incidenty a vytvorím uzly typu incident, ktoré priradím uzlu daného problému.

```
public void initializeNodes () {
// ...
if (problemManageBean.getSelectedTechs() != null && problemManageBean.getSelectedTechs().
    length == 1) {
  problemManageBean.setRoot(new DefaultTreeNode("Root", null));
  TreeNode root = problemManageBean.getRoot();
  ArrayList<DTOProblem> techsProblems = techsToProblemEJB.findProblemsByTechId(
      problemManageBean.getSelectedTechs()[0].getUser_id());
  ArrayList<DTOIncident> problemsIncidents = null;
  for (DTOProblem problem : techsProblems) {
    TreeNode probNode = new DefaultTreeNode("problem", problem, root);
    problemsIncidents = incidentEJB.getRelatedIncidents(problem.getProblem_id());
    for (DTOIncident incident : problemsIncidents) {
      TreeNode incNode = new DefaultTreeNode("incident", incident, probNode);
    }
  }
// ...
}
```

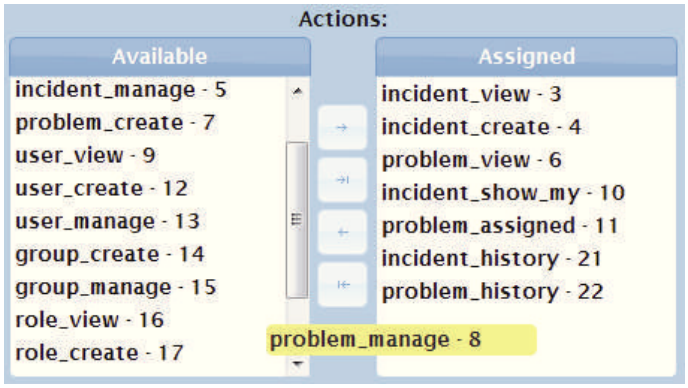
Výpis 12: Tree - inicializácia

Pri odpojení problému od technika získam označený uzol, extrahujem z neho dáta, ktoré pretypujem na problém, ten odstránim z DB a rovnako ho odoberiem aj zo stromu spolu s jeho potomkami.

```
public void unassignProblemFromTech() {
  TreeNode selectedNode = problemManageBean.getSelectedNode();
  Object data = selectedNode.getData();
  techsToProblemEJB.unassignProblemFromTech(problemManageBean.getSelectedTechs()[0].
      getUser_id(), ((DTOProblem) data).getProblem_id());
  selectedNode.getChildren().clear();
  selectedNode.getParent().getChildren().remove(selectedNode);
  selectedNode.setParent(null);
  selectedNode = null;
  problemManageBean.setSelectedNode(selectedNode);
}
```

Výpis 13: Tree - odobratie uzlu

6.1.3 PickList



Obr. 14: Ukážka PickList z aplikácie

`PickList` sa používa na presúvanie dát z jednej kolekcie do druhej. Na to, aby sme ho mohli použiť, musíme implementovať model org.primefaces.model.DualListModel. Tento model pozostáva z dvoch listov, listu zdroja a cieľa. K jednotlivým listom pristupujeme pomocou metód DualListModel.getSource() a DualListModel.getTarget().

POJOs

Na to, aby sme v `PickListe` mohli používať `POJO` musíme implementovať vlastný Converter, ktorý prevedie hodnotu objektu na String a naopak. Converter by mal implementovať javax.faces.convert.Converter a jeho getAsString, getAsObject metódy. Ak by sme chceli zobrazit' zložitejší obsah, ako napr. obrázky môžeme použit' <p:column>.

Nadpisy

Nadpisy pre cieľový a zdrojový list definujeme pomocou facet s menom sourceCaption a targetCaption;

Bližší popis `PickListu` je opät' v príručke v prílohe.

V práci používam `PickList` pri editácii práv rolí.

```
<p:pickList id="actionPickList" value="#{roleManageBean.actions}" var="action" converter="#{
    actionConverter}" itemValue="#{action}" itemLabel="#{action.action_name} - #{action.
    action_id}">
  <f:facet name="sourceCaption">Available</f:facet>
  <f:facet name="targetCaption">Assigned</f:facet>
</p:pickList>
```

Výpis 14: PickList - facelet

Inicializácia DualListModel pre `PickList`:

```
relatedActions = new ArrayList<DTOActions>();
availableActions = new ArrayList<DTOActions>();
actions = new DualListModel<DTOActions>(availableActions, relatedActions);
```

Výpis 15: PickList - inicializácia DualListModel pre PickList

Pred samotnou editáciou je treba naplnit' DualListModel(obsahuje priradené akcie a ostatné dostupné akcie) hodnotami. Zistím aké akcie sú priradené k danej roli a potom podľa toho naplním DualListModel.

```
public void loadActions() {
    DTORole role = roleManageBean.getSelectedRole();
    roleManageBean.setRelatedActions((ArrayList<DTOActions>) role.getActions());
}

public void prepareEdit() {
    // ...
    loadActions();
    roleManageBean.setAvailableActions(roleEJB.getAvailableActions(roleManageBean.
        getSelectedRole()));
    ArrayList<DTOActions> tempAct = (ArrayList<DTOActions>) roleManageBean.
        getRelatedActions().clone();
    roleManageBean.setRelatedActionsForEdit(tempAct);
    roleManageBean.setActions(new DualListModel<DTOActions>(roleManageBean.
        getAvailableActions(),roleManageBean.getRelatedActionsForEdit()));
    // ...
    }
```

Výpis 16: PickList - naplnenie DualListu

6.2 Zaujímavé časti implementácie

6.2.1 Login

Vyriešit' prihlásenie používateľa do aplikácie nie je zložité. Stačí overit', či sa používateľ so zadanými prihlasovacími údajmi nachádza v DB a ak áno, tak inicializujem jeho práva resp. akcie na základe jeho roly a uložím ho do `session`, inak sa zobrazí chybové hlásenie.

```
public String  login () {
    FacesMessage msg = null;
    DTOUser tempUser = loginEJB.userLogin(username);
    if (tempUser != null && passwordMatch(password, tempUser.getUser_password())) {
        loginBean.setUser(tempUser);
        FacesContext.getCurrentInstance().getExternalContext().getSessionMap().put(AUTH_KEY,
            loginBean.getUser());
            if (!loginBean.getUser().isAdmin()) {
                loginBean.getUser().setActionsBasedOnRole(loginEJB.initializeActions(loginBean.
                    getUser()));
            }
        return " restricted /mainInterface.xhtml?faces-redirect=true";
    }
    msg = new FacesMessage(FacesMessage.SEVERITY_WARN, "Login Error", "Invalid
        credentials");
    FacesContext.getCurrentInstance().addMessage(null, msg);
    return null;
}
```

Výpis 17: Ukážka implementácie login metódy

Problém je v tom, že potrebujeme zaručiť, že z jedného účtu môže byť súčasne prihlásený len jeden používateľ. Teda, ak je prihlásený používateľ z účtu X a prihlási sa z tohto účtu ďalší používateľ, potrebujeme tomu prvému zrušiť `session`. Z bezpečnostných dôvodov však nie je umožnené zrušiť `session` z inej `session` (vzdialene). Riešenie tohto problému je, že musíme mať uložené všetky aktívne `sessions` a ich prihlásených používateľov v nejakom **static** atribúte, ktorý je dostupný pre každého používateľa (v každej `session`). Vhodným typom pre reprezentáciu tohto objektu je HashMap, kde bude používateľ vystupovať ako kľúč a jeho `session` ako hodnota. Túto HashMap uchovávame v triede reprezentujúcej používateľa. Teraz potrebujeme pri prihlásení používateľa (vytvorení `session`) zistiť, či sa už nachádza v tejto HashMap alebo nie. Ak sa nachádza, tak ho z nej odstránime a zrušíme jeho uloženú `session` (je to možné pretože k nej pristupujeme lokálne). Pri odhlásení používateľa (zrušení jeho `session`) ho už len stačí odstrániť z HashMap. Potrebujeme teda reagovať na udalosti vytvorenia `session` a zrušenia `session`. Presne to nám umožňujú metódy valueBound(HttpSessionBindingEvent event) a valueUnbound(HttpSessionBindingEvent event) rozhrania HttpSessionBindingListener, pričom valueBound metóda sa volá pri vzniku `session` a naopak metóda valueUnbound pri jej zrušení. Trieda reprezentujúca používateľa teda bude implementovať toto rozhranie, pričom v spomenutých metódach definuje opísanú funkčnosť.

```
public interface DTOUser extends Serializable, Cloneable, HttpSessionBindingListener {
// ...
}
public class User implements DTOUser {
    // ...
    private static  Map<User, HttpSession> logins = new HashMap<User, HttpSession>();
    @Override
    public void valueBound(HttpSessionBindingEvent event) {
        HttpSession session = logins.remove(this);
        if  (session != null)  {
            session.invalidate () ;
        }
        logins.put(this,  event.getSession());
    }
    @Override
    public void valueUnbound(HttpSessionBindingEvent event) {
        logins.remove(this);
    }
    @Override
        public boolean equals(Object other) {
            return (other instanceof User) && (user_name != null) ? user_name.equals(((User) other).
                user_name) : (other == this);
        }
    @Override
    public int  hashCode() {
        return (user_name != null) ? (this.getClass().hashCode() + user_name.hashCode()) : super.
            hashCode();
    }
}
```

Výpis 18: Ukážka implementácie HttpSessionBindingListener

6.2.2 Bezpečnosť

Heslá

Prvá vec, ktorú som ohľadom bezpečnosti riešil bol spôsob uchovávania hesiel v DB. Samozrejme nie je príliš bezpečné uchovávať heslo v takom tvare, v akom ho zadá používateľ, teda napríklad uložiť priamo reťazec „mojeBezpecneHeslo123“ nie je najlepší nápad. Ak by sa v takomto prípade stalo, že potenciálny útočník by získal prístup k DB, poznal by všetky údaje, teda napr. používateľské meno, email, heslo, a iné. Problém je v tom, že takto by nezískal prístup len k účtu používateľa v mojej aplikácii ale taktiež s vysokou pravdepodobnosťou aj k iným službám, ktoré daný používateľ využíva, ako napr. email. Treba si uvedomiť, že je to viac než možné, pretože väčšina ľudí používa rovnaké heslo vo viacerých službách, ak nie vo všetkých.

Heslo preto treba uchovať v takej podobe, aby ho pri prípadnom úniku databáze nemohol útočník zneužiť, či rozšifrovať. K tomuto účelu slúži **hashovanie**. Je to postup pri ktorom sa údaje zašifrujú pomocou **hashovacej funkcie**. Šifrovanie je v tomto prípade **jednosmerné**, čiže zašifrované údaje sa nedajú rozšifrovať, aj keby sme poznali hashovaciu funckiu. Zahashované údaje však tiež nemusia byť vždy bezpečné, napríklad pri použití **MD5** hashovania bez použitia tzv. `salt` (náhodný reťazec pridaný k hashovanému reťazcu pred samotným hashovaním) je možné získať pôvodný reťazec za pomoci tzv. `Rainbow table` (brute force útok). O hashovaní by sa dala napísať celá kapitola avšak to nie je účelom tejto práce. Na záver len uvediem, že na zahashovanie hesiel som zvolil `jBCrypt`, čo je Java implementácia kryptografickej hashovacej funkcie založenej na šifre `Blowfish`. Pre väčšie zabezpečenie by bolo ešte vhodné použiť `HTTPS` pri prihlasovaní. [29]

V návrhu riešenia som uviedol, že je potrebné obmedziť prístupnosť k jednotlivým častiam IS na základe rolí, resp. akcií, ktoré rola obsahuje. Taktiež má byť IS dostupný len prihláseným používateľom. Ponúka sa teda možnosť využiť tzv. `container managed security`, ktorú poskytuje aplikačný server. Pri použití tejto metódy definujeme v `deployment descriptoru web.xml` jednotlivé `roly` a k nim povolené web stránky.[3] Treba však myslieť na to, že v našom prípade rola nemá pevne dané práva (Super admin ich môže meniť) a taktiež môžu byť vytvorené roly úplne nové. Pre tieto špecifické požiadavky je lepšie využiť Filter .

Filter

Požadované oprávnenia (akcie) k jednotlivým web stránkam som uložil do databázovej tabuľky. Uchovávam ich potom v @ApplicationScoped @ManagedBean(eager="true") (inicializuje sa pri štarte aplikácie), čiže sú inicializované len raz a dostupné počas behu celej aplikácie. Vo filtry potom môžeme zabezpečiť prístup do IS len prihláseným používateľom, prístup k jednotlivým web stránkam len používateľom, ktorí majú právo túto stránku zobraziť a presmerovanie používateľa v prípade, že vyprší jeho `session`.

```
// ...
if (session != null) {
    key = session.getAttribute(LoginBean.AUTH_KEY);
    if (!(key instanceof DTOUser)) {
        response.sendRedirect(request.getContextPath() + "/index.xhtml");
    } else if (!ajax && validationRequired(request)) {
            user = (DTOUser) session.getAttribute(LoginBean.AUTH_KEY);
            if (!user.isAdmin()) {
                ServletContext context = request.getServletContext();
                urlActions = (PopulateSupportBean) context.getAttribute("populateBean");
                String url = request.getRequestURL().toString();
                String urlPart = url.substring(url.lastIndexOf('/') + 1);
                if (urlActions.getActionsToUrl().containsKey(urlPart)) {
                    int action_id = urlActions.getActionsToUrl().get(urlPart);
                    if (!user.getActionsBasedOnRole().containsKey(action_id)) {
                        response.sendRedirect("../mainInterface.xhtml");
                    }
                }
            }
            chain.doFilter(req, resp);
        } else {
                chain.doFilter(req, resp);
            }
        } else if (session == null) {
                if (ajax) {
                    response.getWriter().print(xmlAjaxRedirectToPage(request, "/index.xhtml?reason=
                        expired"));
                    response.flushBuffer();
                } else {
                        response.sendRedirect(request.getContextPath() + "/index.xhtml?reason=expired
                            ");
                    }
            }
// ...
```

Výpis 19: Ukážka implementácie hlavnej logiky Filtra

Funkčnosť Filtra je teda nasledujúca : Ak existuje `session`, tak z nej získam používateľa (na základe kľúča definovaného v LoginBean). Ak používateľ nie je **null** a zároveň je inštanciou triedy DTOUser, tak zistím či sa nejedná o `AJAX request` a či je validácia práv potrebná. Pokračujem tým, že získam používateľa zo `session`. Ak používateľ nie je admin, tak načítam jednotlivé url adresy a k nim požadované akcie zo spomínanej eager bean a taktiež zistím url adresu ku ktorej chce používateľ pristúpiť. Ak sa daná url adresa nachádza medzi adresami ku ktorým je potrebná akcia, tak zistím id tejto akcie a overím, či používateľove oprávnenia na základe roly obsahujú túto akciu. Ak sú všetky podmienky splnené, tak je používateľ presmerovaný na stránku ku ktorej chcel pristúpiť. Ak nie sú podmienky splnené, tak je presmerovaný na domovskú stránku.

SQL Injection

Ďalšou potenciálnou hrozbou je **SQL Injection**. Na to, aby som zabránil tomuto typu útoku používam parametrizované dotazy. Problém nastáva u dynamicky generovanej `WHERE` časti pri aplikovaní filtrov z `Primefaces DataTable`. Vyriešil som to tak, že potenciálne nebezpečný znak ' v reťazci získanom z filtra nahradím za '', tým pádom sa nemôže stať, že by sa útočník dostal mimo stringovej časti v DB procedúre a vykonal nejaký SQL dotaz.

6.2.3 Výnimky

V `deployment descriptoru web.xml` sme schopný definovať tzv. <error−page>. Je to stránka na ktorú bude používateľ presmerovaný po tom čo nastane špecifický typ chyby, ktorý sme taktiež definovali.

```
<error−page>
   <exception−type>javax.faces.application.ViewExpiredException</exception−type>
   <location>/error.xhtml</location>
</error−page>
```

Výpis 20: Pridanie error page od web.xml

Tento postup je vcelku jednoduchý, avšak <error−page> nespracúva `AJAX requesty`. Na to, aby sme mohli odchytávať výnimky z `AJAX requestov` musíme vytvoriť vlastnú ExceptionHandlerFactory. Postupujeme nasledovne :

Najskôr treba definovať vlastný ExceptionHandler, ktorý implementuje ExceptionHandlerWrapper. Najdôležitejšia metóda je handle() v ktorej prebieha spracovanie výnimky, viď. ďalšia strana.

```
@Override
    public void handle() throws FacesException {
        Iterable<ExceptionQueuedEvent> events = this.wrapped.
            getUnhandledExceptionQueuedEvents();
        for ( Iterator <ExceptionQueuedEvent> it = events.iterator(); it.hasNext();) {
            ExceptionQueuedEvent event = it.next();
            ExceptionQueuedEventContext eqec = event.getContext();
            if (eqec.getException() instanceof ViewExpiredException) {
                FacesContext context = eqec.getContext();
                NavigationHandler navHandler = context.getApplication().getNavigationHandler();
                try {
                    navHandler.handleNavigation(context, null, "index.xhtml?faces-redirect=true&
                        reason=expired");
                } finally {
                    it.remove();
                }
            } else {
                    FacesContext context = eqec.getContext();
                    NavigationHandler navHandler = context.getApplication().getNavigationHandler();
                    try {
                        logger.addHandler(LogProvider.fh);
                        logger.log(Level.SEVERE, "Uncaugh exception", eqec.getException());
                        navHandler.handleNavigation(context, null, "/error.xhtml?faces-redirect=true");
                    } finally {
                        it.remove();
                    }
            }
        }
        this.wrapped.handle();
    }
```

Výpis 21: ExceptionHandler - metóda handle()

Môžeme tu vykonať potrebné akcie pre každý typ výnimky, ktorý chceme spracovať. Ďalej vytvoríme ExceptionHandlerFactory, ktorá bude používať nami definovaný ExceptionHandler.

```
public class CustomExceptionHandlerFactory extends ExceptionHandlerFactory {
    private ExceptionHandlerFactory base;

    public CustomExceptionHandlerFactory(ExceptionHandlerFactory base) {
        this.base = base;
    }

    @Override
    public ExceptionHandler getExceptionHandler() {
        return new CustomExceptionHandler(base.getExceptionHandler());
    }
}
```

Výpis 22: ExceptionHandlerFactory

Nakoniec našu ExceptionHandlerFactory musíme pridať do `faces-config.xml`, pretože `JSF 2.0` neposkytuje anotáciu na definovanie ExceptionHandlerFactory. [3]

```
<factory>
    <exception-handler-factory>supportClasses.exceptionHandling.
        CustomExceptionHandlerFactory</exception-handler-factory>
</factory>
```

Výpis 23: Definícia ExceptionHandlerFactory vo faces-config.xml

6.3 Problémy a ich riešenia

Počas práce som narazil na viacero problémov. V tejto časti uvediem niektoré z nich.

Vlastná EL funkcia

Jedným z problémov bolo, že ak používateľ zadal príliš dlhý reťaze napr. ako nadpis incidentu, stĺpec tabuľky sa deformoval a prispôsobil dĺžke textu. Riešením bolo, že som si vytvoril vlastnú `EL` funkciu, ktorá dlhý text oreže na základe požiadavky na dĺžku a pridá „...“. Postup vytvorenia funkcie je nasledujúci:

Najskôr vytvoríme **final** triedu, ktorá bude obsahovať **public static** funkciu na orezanie textu.

```
public final class Functions {
    private Functions() {
    }
    public static String ellipsis (String text, int maxLength) {
        text = text.trim();
        return (text.length() > maxLength) ? text.substring(0, maxLength - 1) + "..." : text;
    }
}
```

Výpis 24: Trieda obsahujúca implementáciu EL funkcie

Ďalej musíme v adresári `WEB-INF` vytvoriť `taglib` pre našu funkciu, aby sme ju boli schopní používať z `EL`.

```
<?xml version="1.0" encoding="UTF-8"?>
<facelet-taglib
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
        facelettaglibrary_2_0.xsd"
    version="2.0">
    <namespace>http://itil.bakalarka.com/functions</namespace>

    <function>
        <function-name>ellipsis</function-name>
        <function-class>supportClasses.Functions</function-class>
        <function-signature>String ellipsis(java.lang.String, int)</function-signature>
    </function>
</facelet-taglib>
```

Výpis 25: Definícia vlastnej taglib.xml

Túto `taglib` ešte musíme pridať do `web.xml`.

```
<context-param>
    <param-name>javax.faces.FACELETS_LIBRARIES</param-name>
    <param-value>/WEB-INF/functions.taglib.xml</param-value>
</context-param>
```

Výpis 26: Pridanie taglib do web.xml

Teraz je možno funkciu používať vo `faceletoch`, stačí len definovať nami vytvorený menný priestor.

```
xmlns:func="http://itil.bakalarka.com/functions"

<p:column id="titleColumn" style="width: 150px" headerText="Title" sortBy="#{dataItem.inc_title}"
    filterBy="#{dataItem.inc_title}">
    <h:outputText value="#{func:ellipsis(dataItem.inc_title, 15)}" />
</p:column>
```

Výpis 27: Použitie funkcie z vlastnej taglib vo facelete

Kontrola práv používateľa vo filtry

Pri kontrole práv používateľa vo filtry, by bolo zbytočné tieto práva kontrolovať pre každý `request`. Nie je potrebné kontrolovať práva pre `requesty`, ktoré smerujú na rovnakú stránku z ktorej prišli, pre `requesty` smerujúce na `index.xhtml` a taktiež pre `AJAX requesty`.

Najjednoduchším spôsobom ako zistiť, či sa jedná o `AJAX request` by bolo získať FacesContext a použiť metódu isAjaxRequest() na PartialView. FacesContext však môžeme používať iba u `JSF` komponentov (napr. PhaseListener) a tým Filter nie je. Pre <p:ajax> requesty sa pridáva do hlavičky requestu x-requested-with informácia, že ide o XMLHttpRequest - AJAX request.

```
if (request.getHeader("x-requested-with") != null && request.getHeader("x-requested-with").
    equalsIgnoreCase("XMLHttpRequest")) {
    ajax = true;
}
```

Výpis 28: Zistenie ajax requestu

Zistenie, či `request` smeruje na `index.xhtml` alebo či sa jedná o `request` z rovnakej stránky je potom jednoduché :

```
private boolean validationRequired(HttpServletRequest request) {
    String urlTo = request.getRequestURL().toString();
    String urlFrom = request.getHeader("referer");
    if (urlTo.equals(urlFrom) || request.getContextPath().endsWith("index.xhtml")) {
        return false;
    } else {
        return true;
    }
```

Výpis 29: Overenie requestu

Taktiež musíme zabrániť načítavaniu stránok z `cache` prehliadača. Musia sa načítavať vždy zo serveru, aby bola zaručená aktuálnosť obsahu.

```
if (!request.getRequestURI().startsWith(request.getContextPath() + ResourceHandler.
    RESOURCE_IDENTIFIER)) { // Vynecháme zdroje JSF (CSS/JS/Images/a iné)
  response.setHeader("Cache-Control", "no-cache, no-store, must-revalidate"); // HTTP 1.1.
  response.setHeader("Pragma", "no-cache"); // HTTP 1.0.
  response.setDateHeader("Expires", 0); // Proxies.
}
```

Výpis 30: Zabránenie načítavania stránok z cache

Converter pre akcie

Pri vytváraní Convertera pre akcie som mal problém s načítaním akcií z databáze. Ak som využil pre definovanie Convertera anotáciu @FacesConverter nemohol som injektnúť `EJB` pomocou anotácie @EJB. `EJB` je možné injektovať len v @ManagedBean, takže som bol nútený použiť túto anotáciu a potom definovať Converter v `PickListe` pomocou `EL`:

```
<p:pickList id="actionPickList" value="#{roleManageBean.actions}" converter="#{
    actionConverter}" />
```

Výpis 31: Použitie Convertera ako @ManagedBean v EL

7 Záver

O tvorbu webových aplikácií v Jave, presnejšie o technológiu JSF, som sa z časti zaujímal už pred touto prácou. Počas tvorby práce som však prenikol do tejto problematiky oveľa hlbšie. Získal som mnoho nových znalostí nie len z oblasti Java EE a JSF 2.0, ale aj iných technológií a frameworkov, ako napr. jQuery, AJAX, CSS, či HTML. Taktiež som sa zoznámil so základmi šifrovania a bezpečnosti webových aplikácií. Najväčším prínosom však pre mňa bolo osvojenie si znalostí ohľadom frameworku Primefaces. Rád by som sa štúdiu Primefaces (a JSF 2.0) venoval aj naďalej, pretože poskytuje naozaj bohaté portfólio komponentov a v súčasnosti sa jedná o jeden z najžiadanejších JSF 2.0 frameworkov na trhu. Problémy (a ich riešenia), na ktoré som narazil počas tvorby tejto práce ma taktiež motivovali k tomu, aby som začal aktívne prispievať svojimi vedomosťami na stránkach Stackoverflow[4].

Do budúcna mi napadá množstvo vylepšení, ktoré by som chcel do aplikácie zakomponovať, napr. detailné spracovanie štatistík pre jednotlivé skupiny, ako počty používateľov, incidentov, problémov, prehľad vzniku incidentov v čase a pod. Štatistiky budú spracované v tabuľkovej forme ale taktiež vo forme grafov, pre väčšiu prehľadnosť. Ďalej by som rád vylepšil spracúvanie filtrov tabuliek, zamenil čisté JDBC za JPA, implementoval tlstého klienta s pomocou webových služieb a pod. Treba tiež doplniť lokalizáciu. Žiadna aplikácia nie je dokonalá a vždy je čo vylepšovať, preto by som rád vo vývoji pokračoval a využil túto aplikáciu aj ako základ pre moju diplomovú prácu.

Konštatujem, že požiadavky zadania boli splnené, počas práce som vyriešil všetky väčšie komplikácie a prehĺbil si vedomosti v oblasti viacerých technológií. Prácu teda pre mňa hodnotím ako veľmi prínosnú.

Na záver by som chcel povedať, že podľa môjho názoru má Primefaces a technológia JSF všeobecne pred sebou ešte veľkú budúcnosť, pretože webové aplikácie zohrávajú a určite aj budú zohrávať dôležitú úlohu, či už v oblasti automatizácie biznis procesov alebo inej.

Martin Gajdičiar

[4] Jedná sa o stránku kde si programátori vzájomne pomáhajú pri riešení rôznych problémov

8 Literatúra

[1] Jacobi, J. - Fallows R., J. *Pro JSF and Ajax: Building Rich Internet Components* Berkeley CA: Apress, 2006. ISBN 1-59059-580-7

[2] Mann D., K. *JavaServer Faces in Action* Greenwich: Manning Publications Co., 2005. ISBN 1-932394-11-7

[3] Anghel, L. *JSF 2.0 Cookbook* Birmingham: Packt Publishing Ltd., 2010. ISBN 978-1-847199-52-2

[4] Bien, A. *Real World Java EE Patterns - Rethinking Best Practices* Birmingham: press.adam-bien.com, 2009. ISBN 978-0-557-07832-5

[5] *Distributed Multitiered Applications* [online]. c2012, [cit. 2012-04-14]. URL:
<http://docs.oracle.com/javaee/6/tutorial/doc/bnaay.html>

[6] *JavaServer Faces Technology* [online]. c2012, [cit. 2012-04-14].URL:
<http://docs.oracle.com/javaee/6/tutorial/doc/bnaph.html>

[7] *JSF 2.0 Views: Hello Facelets, Goodbye JSP* [online]. 2010-03-01, [cit. 2012-04-14].URL:
<http://www.developer.com/java/web/article.php/3867851/JSF-20-Views-Hello-Facelets-Goodbye-JSP.htm>

[8] *Java Servlet Technology* [online]. c2012, [cit. 2012-04-14].URL:
<http://docs.oracle.com/javaee/6/tutorial/doc/bnafe.html>

[9] *Filtering Requests and Responses* [online]. c2012, [cit. 2012-04-14].URL:
<http://docs.oracle.com/javaee/6/tutorial/doc/bnagb.html>

[10] *Enterprise Beans* [online]. c2012, [cit. 2012-04-14].URL:
<http://docs.oracle.com/javaee/6/tutorial/doc/gipmb.html>

[11] *Session Beans* [online]. c2012, [cit. 2012-04-14].URL:
<http://docs.oracle.com/javaee/6/tutorial/doc/gipjg.html>

[12] *Message-Driven Beans* [online]. c2012, [cit. 2012-04-14].URL:
<http://docs.oracle.com/javaee/6/tutorial/doc/gipko.html>

[13] *Java EE 6 APIs in the Java Platform, Standard Edition 6.0* [online]. c2012, [cit. 2012-04-14].URL:
<http://docs.oracle.com/javaee/6/tutorial/doc/girdr.html>

[14] *Introducing the Java EE 6 Platform: Part 3* [online]. 2009-12-12, [cit. 2012-04-14].URL:
<http://java.sun.com/developer/technicalArticles/JavaEE/JavaEE6Overview_Part3.html>

[15] Scholtz,B *Communication in JSF 2.0* [online]. 2011-09-16, [cit. 2012-04-15].URL:
<http://balusc.blogspot.com/2011/09/communication-in-jsf-20.html>

[16] *Making distinctions between different kinds of JSF Managed Beans* [online]. 2009-04-23, [cit. 2012-04-15].URL:
<http://blog.icesoft.org/blojsom/blog/default/2009/04/23/Making-distinctions-between-different-kinds-of-JSF-managed-beans/>

[17] Hightower,R *JSF for nonbelievers: The JSF application lifecycle* [online]. 2005-03-01, [cit. 2012-04-16].URL:
<http://www.ibm.com/developerworks/library/j-jsf2/>

[18] *The Lifecycle of a JavaServer Faces Application* [online]. c2012, [cit. 2012-04-16].URL:
<http://docs.oracle.com/javaee/6/tutorial/doc/bnaqq.html>

[19] *The Life Cycle of a JavaServer Faces Page* [online]. 2007-2010, [cit. 2012-04-16].URL:
<http://docs.oracle.com/javaee/5/tutorial/doc/bnaqq.html>

[20] *Why primefaces* [online]. c2011, [cit. 2012-04-16].URL:
<http://primefaces.org/whyprimefaces.html>

[21] Skála,J *Čo je to ITSM* [online]. 2008-2012, [cit. 2012-04-17].URL:
<http://www.itsm.sk/sk/-ITSM-ITIL/Co-je-to-ITSM.alej>

[22] Skála,J *Čo je to služba IT* [online]. 2008-2012, [cit. 2012-04-17].URL:
<http://www.itsm.sk/sk/-ITSM-ITIL/Co-je-to-sluzba-IT.alej>

[23] Skála,J *Čo je to ITIL* [online]. 2008-2012, [cit. 2012-04-17].URL:
<http://www.itsm.sk/sk/-ITSM-ITIL/Co-je-to-ITIL-.alej>

[24] Skála,J *Čo (ne)môžeme od ITIL očakávať* [online]. 2008-2012, [cit. 2012-04-17].URL:
<http://www.itsm.sk/sk/-ITSM-ITIL/Co-ne-mozeme-od-ITIL-ocakavat.alej>

[25] Skála,J *Incident management* [online]. 2008-2012, [cit. 2012-04-17].URL:
<http://www.itsm.sk/sk/-ITSM-ITIL/Klucove-procesy-ITIL-V3/Incident-management.alej$>$

[26] Skála,J *Problem management* [online]. 2008-2012, [cit. 2012-04-17].URL:
<http://www.itsm.sk/sk/-ITSM-ITIL/Klucove-procesy-ITIL-V3/Problem-management.alej>

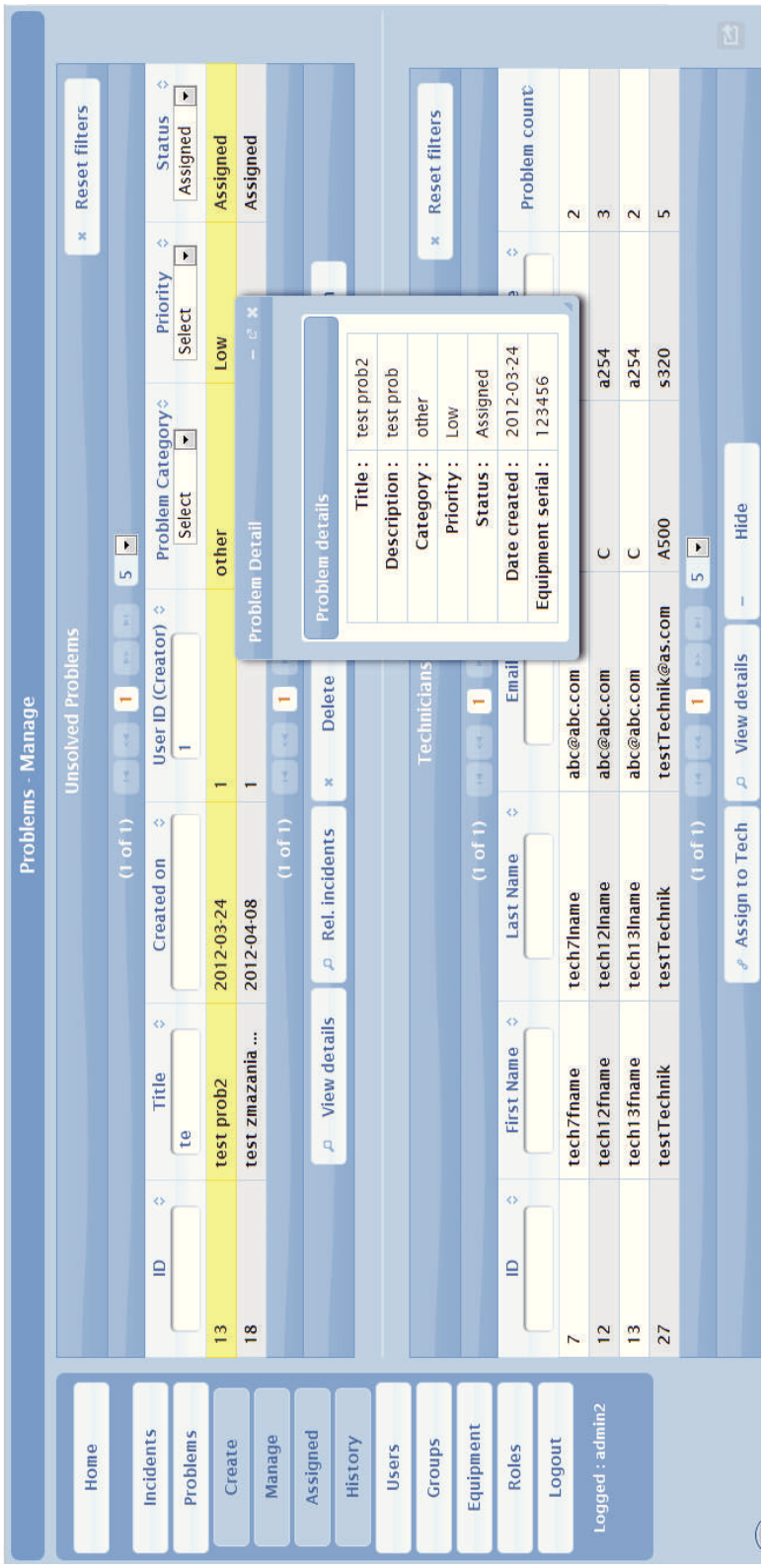
[27] Skála,J *Service desk* [online]. 2008-2012, [cit. 2012-04-17].URL:
<http://www.itsm.sk/sk/-ITSM-ITIL/Funkce-ITIL-V3/Service-desk.alej>

[28] *Core J2EE Patterns - Data Access Object* [online]. 2003-04-12, [cit. 2012-04-18].URL:
<http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>

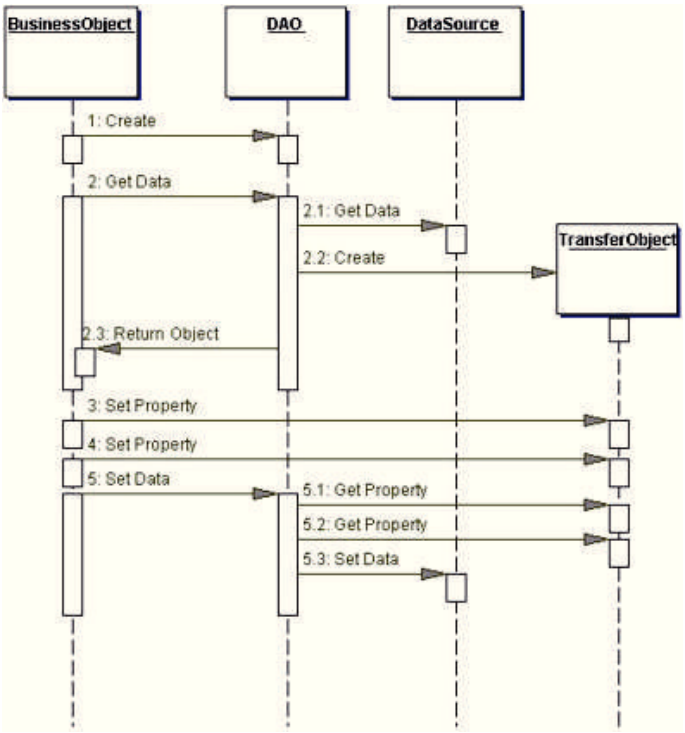
[29] Ptacek,T *Enough With The Rainbow Tables: What You Need To Know About Secure Password Schemes* [online]. 2007-09-07, [cit. 2012-04-19].URL:
<http://chargen.matasano.com/chargen/2007/9/7/enough-with-the-rainbow-tables-what-you-need-to-know-about-s.html>

A Doplňujúce obrázky a diagramy

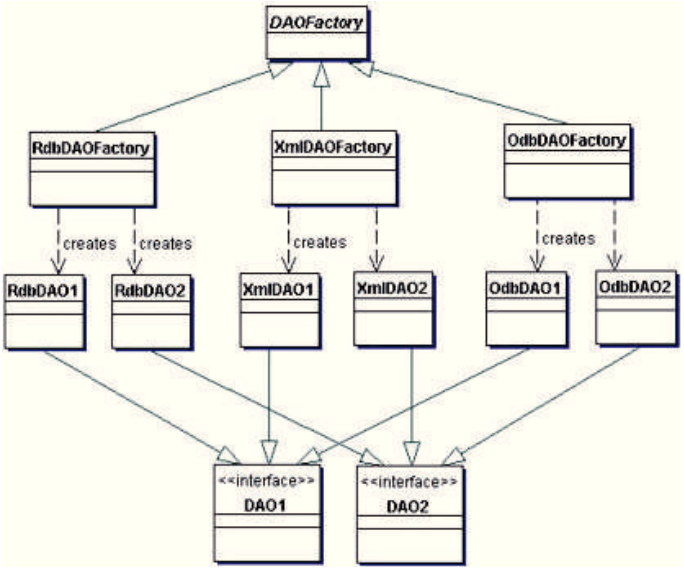
Obrázky, ktoré by svojou veľkosťou prekážali v texte.



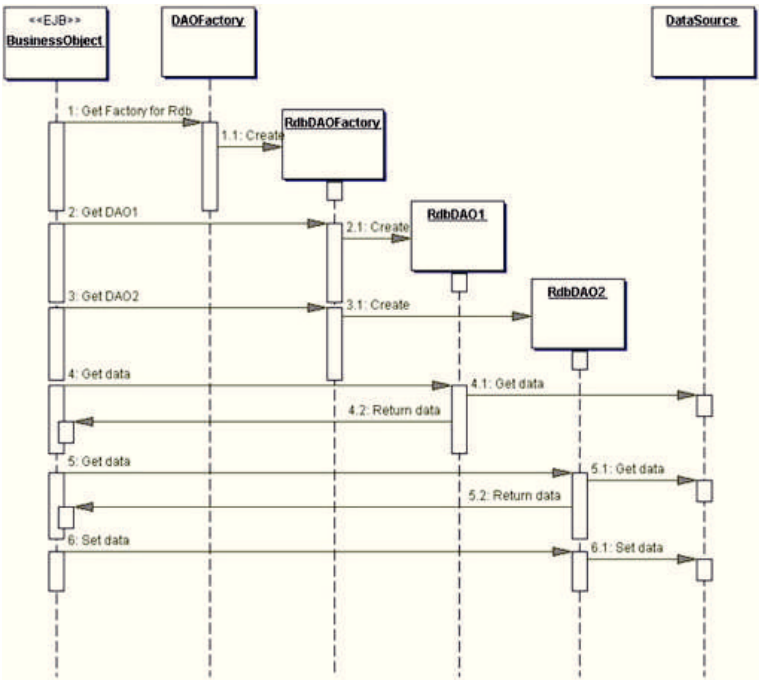
Obr. 15: Ukážka GUI systému



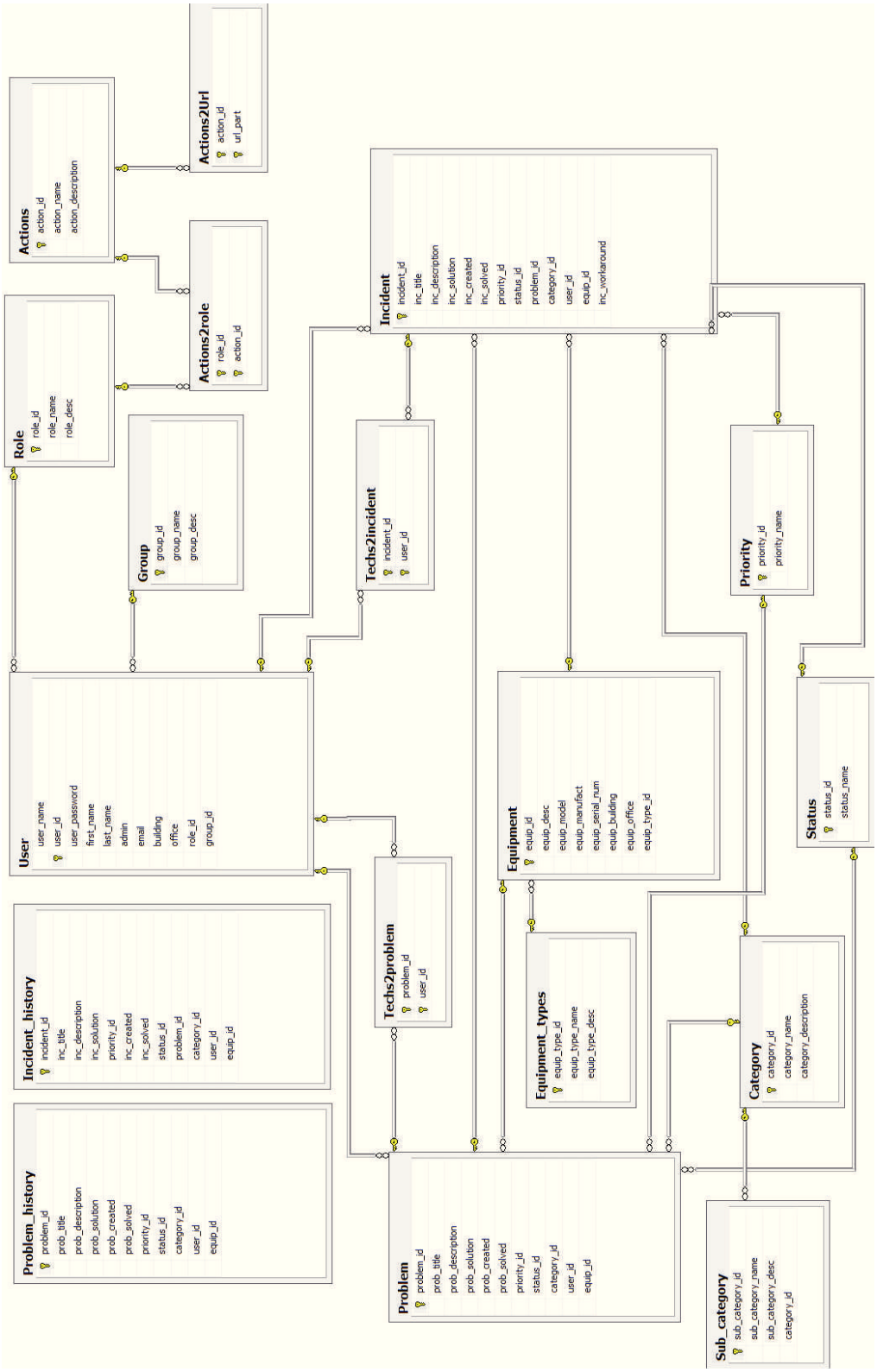
Obr. 16: Sekvenčný diagram pre DAO Factory



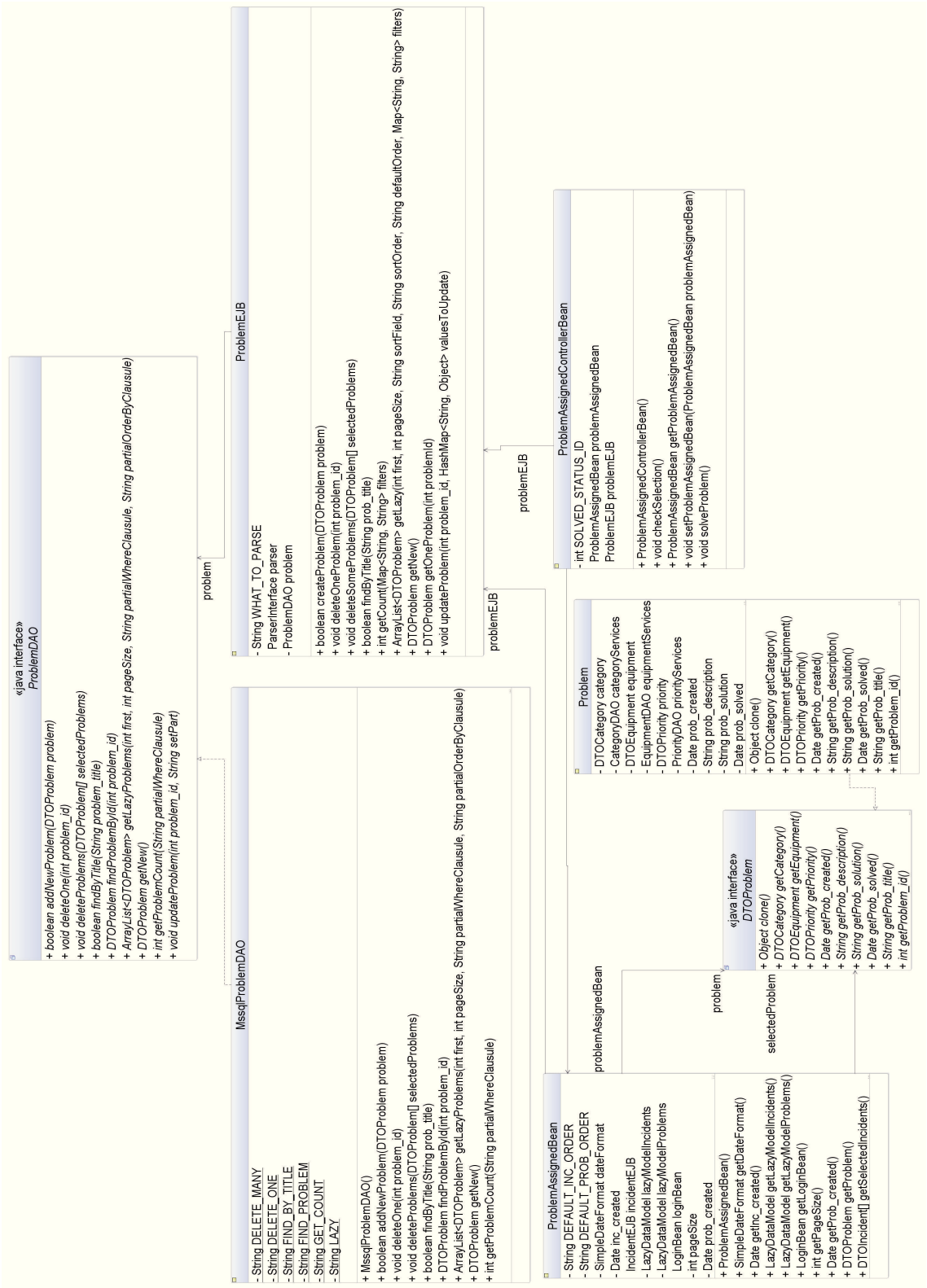
Obr. 17: Triedny diagram Abstract DAO Factory



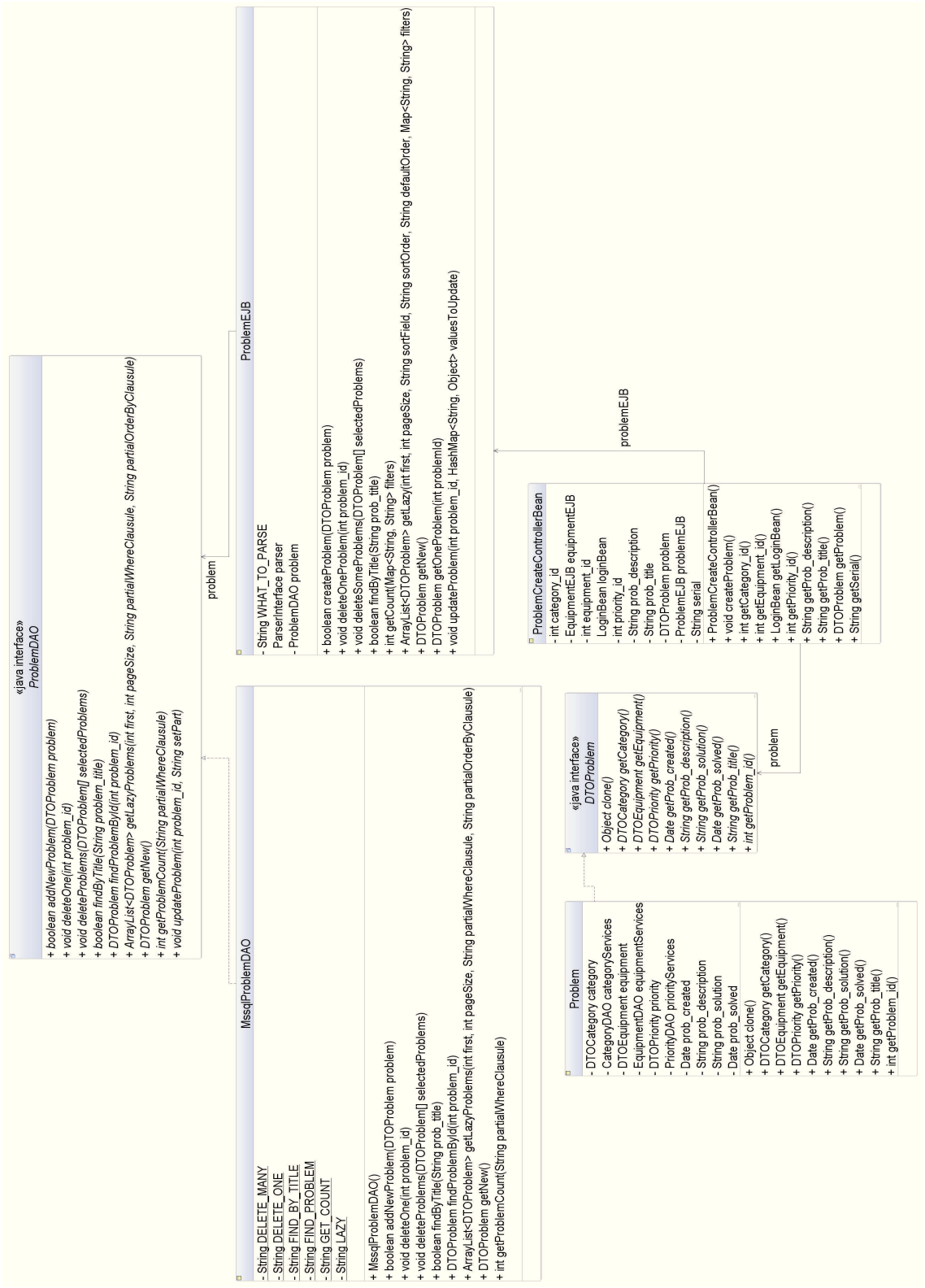
Obr. 18: Sekvenčný diagram Abstract DAO Factory



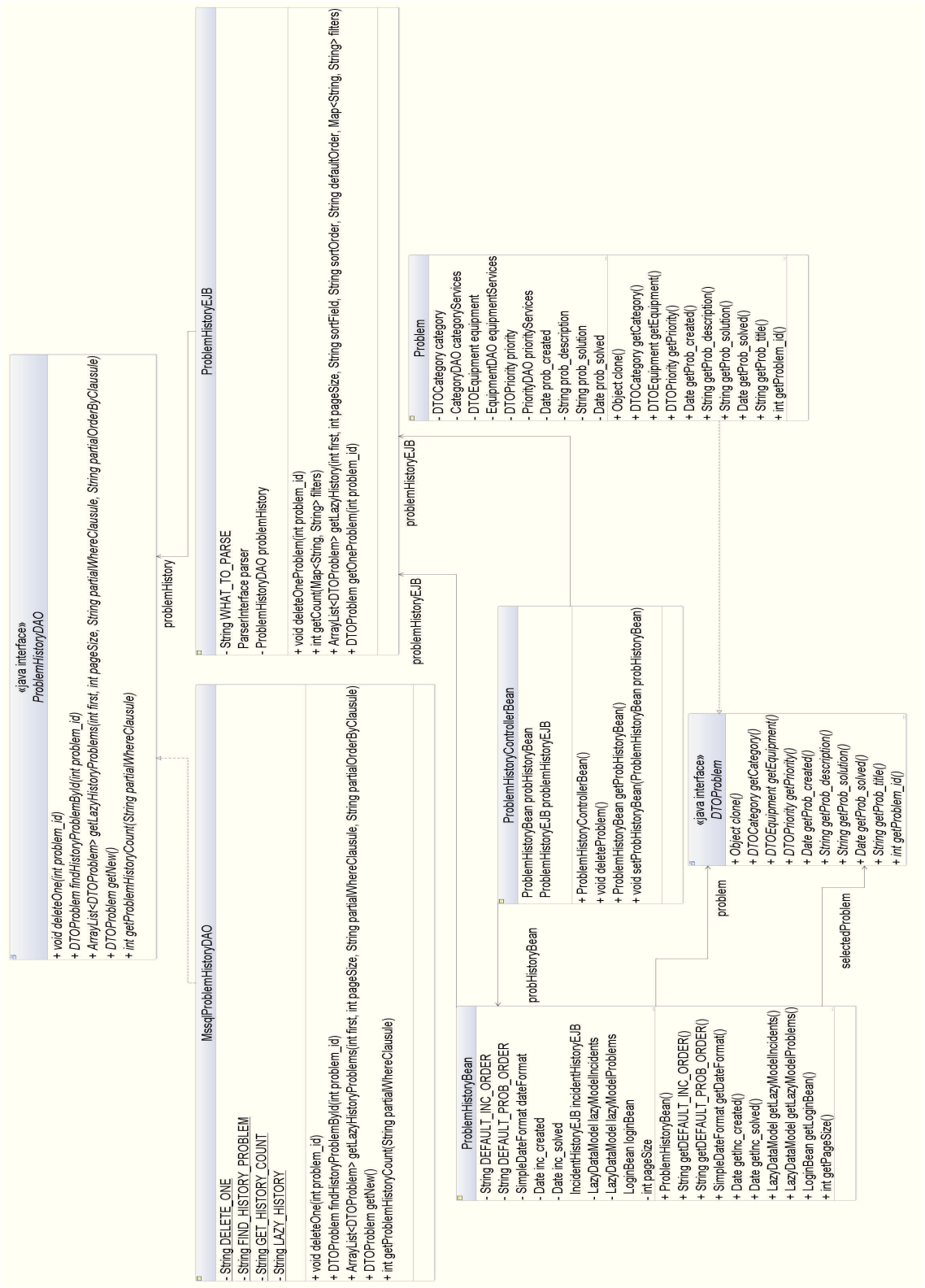
Obr. 19: ER Diagram



Obr. 20: Java class diagram: Priradené problémy - zjednodušený



Obr. 21: Java class diagram: Vytvorenie problému - zjednodušený



Obr. 22: Java class diagram: História problémov - zjednodušený

B Obsah priloženého CD

CD obsahuje priečinky:

- BPText - Text bakalárskej práce vo formáte PDF
- SQLScripts - SQL skripty potrebné na vytvorenie DB tabuliek, procedúr a triggerov
- PromoVid - Krátke prezentačné video s ukážkovou funkčnosťou systému
- ProjectNB - NetBeans IDE projekt aplikácie
- OverviewRolesModules - prehľad všetkých akcií rolí a jednotlivých modulov
- Diagrams - rôzne diagramy (Use case, triedne, ...)
- UserGuide - používateľská príručka pre Primefaces
- Screenshots - screenshoty systému, prierez vývojom
- AdditionalLibraries - potrebné knižnice k aplikácii